IBM LoadLeveler
Version 5 Release 1

Using and Administering

SC23-6792-04

IBM LoadLeveler
Version 5 Release 1

Using and Administering

SC23-6792-04

Note
Before using this information and the product it supports, read the information in "Notices" on page 423.

This edition applies to version 5, release 1, modification 0 of IBM LoadLeveler (product numbers 5725-G01, 5641-LL1, 5641-LL3, 5765-L50, and 5765-LLP) and to all subsequent releases and modifications until otherwise indicated in new editions.

This edition replaces SC23-6792-03.

Contents

Figures

Tables

About this information

Attention:

For LoadLeveler® Version 5 Release 1, changes apply to Linux.

Disclaimer:

The functions or features found herein may not be available on all operating systems or platforms and do not indicate the availability of these functions or features within the IBM® product or future versions of the IBM product. The development, release, and timing of any future features or functionality is at IBM's sole discretion. IBM's plans, directions, and intent are subject to change or withdrawal without notice at IBM's sole discretion. The information mentioned is not a commitment, promise, or legal obligation to deliver any material, code or functionality. The information may not be incorporated into any contract and it should not be relied on in making a purchasing decision.

Release notes:

For the latest release notes, go to the Fix Central website (http://www-933.ibm.com/support/fixcentral/?productGroup0=ibm/fcpower&productGroup1=ibm/ClusterSoftware&productGroup2=ibm/power/IBM). For more information about Fix Central, see "Locating LoadLeveler software fixes and updates in Fix Central" on page xiii.

IBM LoadLeveler provides various ways of scheduling and managing applications for best performance and most efficient use of resources. LoadLeveler manages both serial and parallel jobs over a cluster of machines or servers, which may be desktop workstations, dedicated servers, or parallel machines. This information describes how to configure and administer this cluster environment, and to submit and manage jobs that run on machines in the cluster.

Who should use this information

This book is intended for two separate audiences:

- Personnel who are responsible for installing, configuring and managing the LoadLeveler cluster environment. These people are called LoadLeveler administrators. LoadLeveler administrative tasks include:
  - Setting up configuration and administration files
  - Maintaining the LoadLeveler product
  - Setting up the distributed environment for allocating batch jobs
- Users who submit and manage serial and parallel jobs to run in the LoadLeveler cluster.

Both LoadLeveler administrators and general users should be experienced with the UNIX commands. Administrators also should be familiar with:

- Cluster system management techniques such as SMIT, as it is used in the AIX® environment
- Networking and NFS or AFS® protocols

Conventions and terminology used in this information

Throughout the IBM LoadLeveler product information:

- LoadLeveler for Linux on x86 Architecture includes:
  - IBM System x® Intelligent Cluster™

  - IBM System servers with Advanced Micro Devices (AMD) Opteron or Intel® Extended Memory 64 Technology (EM64T) processors
- References to Schedd are also referred to as job manager.

Table 1. Conventions

Convention	Usage
bold	**Bold** words or characters represent system elements that you must use literally, such as commands, flags, path names, directories, file names, values, and selected menu options.
bold underlined	**Bold underlined** keywords are defaults. These take effect if you do not specify a different keyword.
`constant width`	Examples and information that the system displays appear in `constant-width` typeface.
italic	*Italic* words or characters represent variable values that you must supply. *Italics* are also used for information unit titles, for the first use of a glossary term, and for general emphasis in text.
<*key*>	Angle brackets (less-than and greater-than) enclose the name of a key on the keyboard. For example, <**Enter**> refers to the key on your terminal or workstation that is labeled with the word *Enter*.

\	In command examples, a backslash indicates that the command or coding example continues on the next line. For example:
	`mkcondition -r IBM.FileSystem -e "PercentTotUsed > 90" \`
	`-E "PercentTotUsed < 85" -m d "FileSystem space used"`

{*item*}	Braces enclose a list from which you must choose an item in format and syntax descriptions.
[*item*]	Brackets enclose optional items in format and syntax descriptions.
<**Ctrl-***x*>	The notation <**Ctrl-***x*> indicates a control character sequence. For example, <**Ctrl-c**> means that you hold down the control key while pressing <**c**>.
item...	Ellipses indicate that you can repeat the preceding item one or more times.
|	• In *syntax* statements, vertical lines separate a list of choices. In other words, a vertical line means *Or*. • In the left margin of the document, vertical lines indicate technical changes to the information.

Prerequisite and related information

The LoadLeveler publications are:

- *AIX Installation Guide*, SC23-6791
- *Linux Installation Guide*, SC23-6789
- *Using and Administering*, SC23-6792
- *Diagnosis and Messages Guide*, SC23-6793
- *Command and API Reference*, SC23-6794
- *Resource Manager*, SC23-6790

To access all LoadLeveler documentation, refer to the IBM Cluster Information Center (http://publib.boulder.ibm.com/infocenter/clresctr/vxrx/index.jsp), which contains the most recent LoadLeveler documentation in PDF and HTML formats.

A **LoadLeveler Documentation Updates** file also is maintained on this Web site. The **LoadLeveler Documentation Updates** file contains updates to the LoadLeveler documentation. These updates include documentation corrections and clarifications that were discovered after the LoadLeveler information units were published.

Both the current LoadLeveler books and earlier versions of the library are also available in PDF format from the IBM Publications Center (http://www.ibm.com/e-business/linkweb/publications/servlet/pbi.wss).

To easily locate a book in the IBM Publications Center, supply the book's publication number. The publication number for each of the LoadLeveler books is listed after the book title in the preceding list.

Locating LoadLeveler software fixes and updates in Fix Central

To locate LoadLeveler software fixes and updates, do the following:

1. Go to the Fix Central website (http://www-933.ibm.com/support/fixcentral/?productGroup0=ibm/fcpower&productGroup1=ibm/ClusterSoftware&productGroup2=ibm/power/IBM).
2. In the **Product Group** pull-down menu, under **Software**, select **Cluster software**.
3. In the **Select from Cluster software** pull-down menu, choose **LoadLeveler**.
4. In the **Installed Version** pull-down menu, select the version you have installed, for example, **5.1.0**.
5. In the **Platform** pull-down menu, select the platform, for example, **All**.
6. Click **Continue** and select an option on the **Identify fixes** panel. For example, select **Browse for fixes** and click **Continue** again.
7. On the **Select fixes panel**, select fixes to download or, to view the Restrictions list, click on **Readme**, next to the appropriate fix and open the **Known Limitations** section.

How to send your comments

Your feedback is important in helping us to produce accurate, high-quality information. If you have any comments about this book or any other LoadLeveler documentation, send your comments by e-mail to:

mhvrcfs@us.ibm.com

Include the book title and order number, and, if applicable, the specific location of the information you have comments on (for example, a page number or a table number).

For technical information and to exchange ideas related to high performance computing, go to:

- HPC Central (http://www.ibm.com/developerworks/wikis/display/hpccentral/HPC+Central)
- HPC Central Technical Forum (http://www.ibm.com/developerworks/forums/forum.jspa?forumID=1056)

Summary of changes

Version 5 Release 1

The following sections summarize changes to the IBM LoadLeveler product and library.

Within each information unit in the library, a vertical line to the left of text and illustrations indicates technical changes or additions made to the previous edition of the information.

Attention:

For LoadLeveler Version 5 Release 1, changes apply to Linux.

Disclaimer:

The functions or features found herein may not be available on all operating systems or platforms and do not indicate the availability of these functions or features within the IBM product or future versions of the IBM product. The development, release, and timing of any future features or functionality is at IBM's sole discretion. IBM's plans, directions, and intent are subject to change or withdrawal without notice at IBM's sole discretion. The information mentioned is not a commitment, promise, or legal obligation to deliver any material, code or functionality. The information may not be incorporated into any contract and it should not be relied on in making a purchasing decision.

Release notes:

For the latest release notes, go to the Fix Central website (http://www-933.ibm.com/support/fixcentral/?productGroup0=ibm/fcpower&productGroup1=ibm/ClusterSoftware&productGroup2=ibm/power/IBM). For more information about Fix Central, see "Locating LoadLeveler software fixes and updates in Fix Central" on page xiii.

Changes to LoadLeveler for Linux Version 5 Release 1 product and library include:

- **New information:**
  - A new job command file keyword, **first_node_tasks**, has been added to allow users to request a different task count for the first node. The value specified for **tasks_per_node** would apply to all nodes except the first node.
  - Support for LoadLeveler as the scheduler for Blue Gene®/Q has been added.
    - A new Blue Gene® command called **llbgctl** has been added. You can use the **llbgctl** command to control Blue Gene system resources.
    - A new command called **llbgstatus** has been added that returns Blue Gene system status information.

    For more information, see *LoadLeveler: Command and API Reference*.
  - LoadLeveler now permits its **execute** directory to be located in volatile storage, such as RAM disk, where the contents may not exist following the reboot of an execute node. To avoid the potential performance impact of using a shared file system, configure the **execute** directory in either local disk, or in RAM disk.
  - Support for Workload Manager (WLM) on LoadLeveler for Linux on POWER® (RHEL 6.2) and on x86_64 (SLES 11) has been added.
  - A new affinity interface is introduced into LoadLeveler. The new interface is used by IBM Parallel Environment (PE) Runtime Edition on the x86 platform.

- Island scheduling is introduced in LoadLeveler. New configuration parameters permit administrators to define islands within a LoadLeveler cluster. You can now request that your jobs run within one island or across multiple islands.
- Support for running Intel Message Passing Interface (MPI) and Open MPI jobs as batch LoadLeveler jobs has been added.
- The **llrun** command has been added to allow you to run an interactive command that uses resources allocated by LoadLeveler.
- LoadLeveler provides support for running embarrassingly parallel jobs without using an MPI runtime process manager. For more information, see the following:
  - *LoadLeveler: Command and API Reference*
  - *LoadLeveler: Using and Administering*
- Support for the energy function on the x86_64 iDataPlex® Sandy Bridge processor for SUSE Linux Enterprise Server (SLES) 11 includes the following:
  - The ability to run jobs in a lower CPU frequency. To minimize energy consumption with minimal performance degradation, LoadLeveler provides the job energy policy tag to allow you to set the acceptable performance degradation for the job.
  - Job energy reports can be generated from the accounting data by using the **llsummary** command. The job energy consumption is calculated after the job finishes.
  - An administrator can switch idle nodes to standby state to save energy. This can be done either manually or LoadLeveler can do it automatically. You can wake up the idle nodes that are in standby mode to run the workload as needed.

  To support the energy function, the following have been added:
  - Four new commands: **llrchgmstat**, **llreinit**, **llrqetag**, **llrrmetag**. See *LoadLeveler: Resource Manager* for more information about these commands.
  - Two new subroutines: **llr_change_node_powerstate** and **llr_remove_energy_tags**. See *LoadLeveler: Resource Manager* for more information about these subroutines.

  For additional information, see the following:
  - *LoadLeveler: Command and API Reference*
  - *LoadLeveler: Using and Administering*
  - *LoadLeveler for Linux: Installation Guide*
- Support for checkpoint/restart on Red Hat Enterprise Linux (RHEL) 6.2, which includes the following:
  - The system administrator or user can now set the checkpoint interval on a per job basis.
  - Task migration, which provides a method to move portions of a running parallel job from the current allocated set of nodes to a new set of nodes. To support task migration, the following have been added:
    - The **llmigrate** command
    - The **ll_migrate_job_step** subroutine

  For more information, see the following:
  - *LoadLeveler: Command and API Reference*
  - *LoadLeveler: Resource Manager*
  - *LoadLeveler: Using and Administering*

- Support for multiple endpoints has been added. The new **endpoints** keyword is supported on a class basis for all jobs submitted in that class, as well as an option on the **network** keyword. For more information, see the following:
  - *LoadLeveler: Using and Administering*
  - *LoadLeveler: Resource Manager*
- Support for running LoadLeveler on Blue Gene/Q has been added on LoadLeveler for Linux on POWER (RHEL 6.2). For more information, see the following:
  - *LoadLeveler: Command and API Reference*
  - *LoadLeveler: Using and Administering*
  - *LoadLeveler: Resource Manager*

- **Changed information:**
  - The **llrctl** command is changed to be consistent with the LoadLeveler scheduler command, **llctl**. The same keywords supported in the **llctl** command are now supported in the **llrctl** command. The **drain jobmgr** and **resume jobmgr** keywords are no longer supported in the **llrctl** command.
  - The following table summarizes the Blue Gene/Q terminology changes:

BG/L and BG/P (previous terms)	BG/Q terms
Partitions	Blocks
Base partitions	Midplanes
Wires	Cables
Node cards	Node boards
mpirun, submit, mpiexec	runjob

- **Deleted information:**
  - Support for using virtual IP (VIP) addresses with checkpoint or restart has been removed, so all references to VIPs or the VIP server have been removed from LoadLeveler publications.
  - The **ALLOC_EXCLUSIVE_CPU_PER_JOB** configuration file keyword has been deprecated and has been removed.
  - The following Blue Gene job command file keywords have been deprecated and removed:
    - **bg_connection** (replaced with **bg_connectivity**)
    - **bg_partition** (replaced with **bg_block**)

Part 1. Overview of LoadLeveler concepts and operation

Setting up IBM LoadLeveler involves defining machines, users, jobs, and how they interact, in such a way that LoadLeveler is able to run jobs quickly and efficiently.

Once you have a basic understanding of the LoadLeveler product and its interfaces, you can find more details in the topics listed in Table 2.

Table 2. Major topics in LoadLeveler: Using and Administering

To learn about:	**Read the following:**
Performing administrator tasks	Part 2, "Configuring and managing the LoadLeveler environment," on page 37
Performing general user tasks	Part 3, "Submitting and managing LoadLeveler jobs," on page 169
Using LoadLeveler interfaces	Part 4, "LoadLeveler interfaces reference," on page 229

Chapter 1. What is LoadLeveler?

LoadLeveler is a job management system that allows users to run more jobs in less time by matching the jobs' processing needs with the available resources. LoadLeveler schedules jobs, and provides functions for building, submitting, and processing jobs quickly and efficiently in a dynamic environment.

Figure 1 shows the different environments to which LoadLeveler can schedule jobs. Together, these environments comprise the *LoadLeveler cluster*.

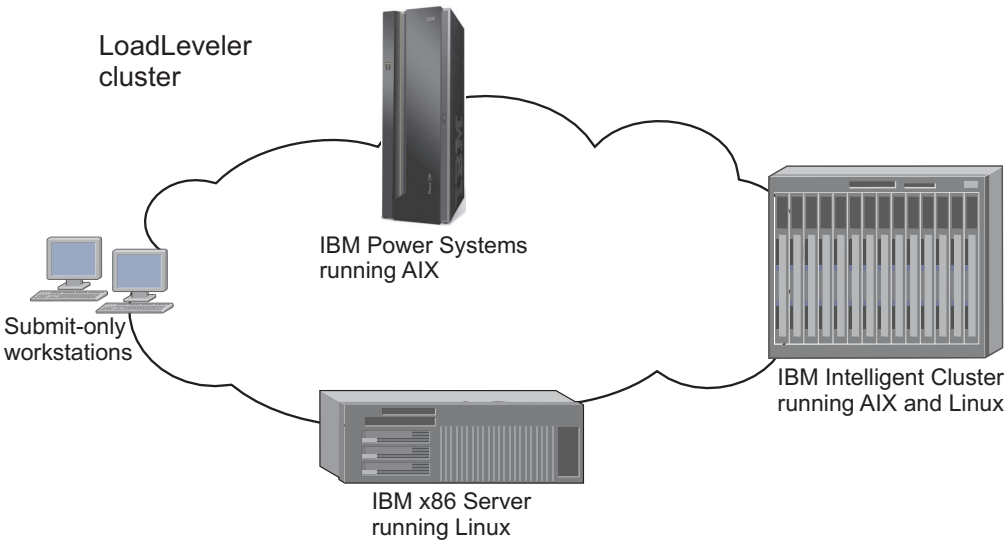

Figure 1. Example of a LoadLeveler cluster

As Figure 1 also illustrates, a LoadLeveler cluster can include *submit-only* machines, which allow users to have access to a limited number of LoadLeveler features.

Throughout all the topics, the terms *workstation*, *machine*, *node*, and *operating system instance (OSI)* refer to the machines in your cluster. In LoadLeveler, an OSI is treated as a single instance of an operating system image.

If you are unfamiliar with the LoadLeveler product, consider reading one or more of the introductory topics listed in Table 3:

Table 3. Topics in the LoadLeveler overview

To learn about:	Read the following:
Using the default configuration for getting a quick start	Chapter 2, "Getting a quick start using the default configuration," on page 29
Specific products and features that are required for or available through the LoadLeveler environment	Chapter 3, "What operating systems are supported by LoadLeveler?," on page 35

LoadLeveler basics

LoadLeveler has various types of interfaces that enable users to create and submit jobs and allow system administrators to configure the system and control running jobs.

These interfaces include:

- Control files that define the elements, characteristics, and policies of LoadLeveler and the jobs it manages. These files are the configuration file, the administration file, and job command file. Optionally, a configuration database can replace the configuration and administration files.
- The command line interface, which gives you access to basic job and administrative functions.
- An application programming interface (API), which allows application programs written by users and administrators to interact with the LoadLeveler environment.

The commands and APIs permit different levels of access to administrators and users. User access is typically restricted to submitting and managing individual jobs, while administrative access allows setting up system configurations, job scheduling, and accounting.

Users create job command files that instruct the system on how to process information. Each job command file consists of keywords followed by the user defined association for that keyword. For example, the keyword **executable** tells LoadLeveler that you are about to define the name of a program you want to run. Therefore, **executable = longjob** tells LoadLeveler to run the program called *longjob*.

After creating the job command file, you invoke LoadLeveler commands to monitor and control the job as it moves through the system. LoadLeveler monitors each job as it moves through the system using process control daemons. However, the administrator maintains ultimate control over all LoadLeveler jobs by defining job classes that control how and when LoadLeveler will run a job.

In addition to setting up job classes, the administrator can also control how jobs move through the system by specifying the type of scheduler. LoadLeveler has several different scheduler options that start jobs using specific algorithms to balance job priority with available machine resources.

When LoadLeveler administrators are configuring clusters and when users are planning jobs, they need to be aware of the machine resources available in the cluster. These resources include items like the number of CPUs and the amount of memory available for each job. Because resource availability will vary over time, LoadLeveler defines them as consumable resources.

LoadLeveler: A network job management and scheduling system

A network job management and job scheduling system, such as LoadLeveler, is a software program that schedules and manages jobs that you submit to one or more machines under its control.

LoadLeveler accepts jobs that users submit and reviews the job requirements. LoadLeveler then examines the machines under its control to determine which machines are best suited to run each job.

Job definition

LoadLeveler schedules your jobs on one or more machines for processing. The definition of a **job**, in this context, is a set of **job steps**. For each job step, you can specify a different executable (the executable is the part of the job that gets processed). You can use LoadLeveler to submit jobs which are made up of one or more job steps, where each job step depends upon the completion status of a previous job step. For example, Figure 2 illustrates a stream of job steps:

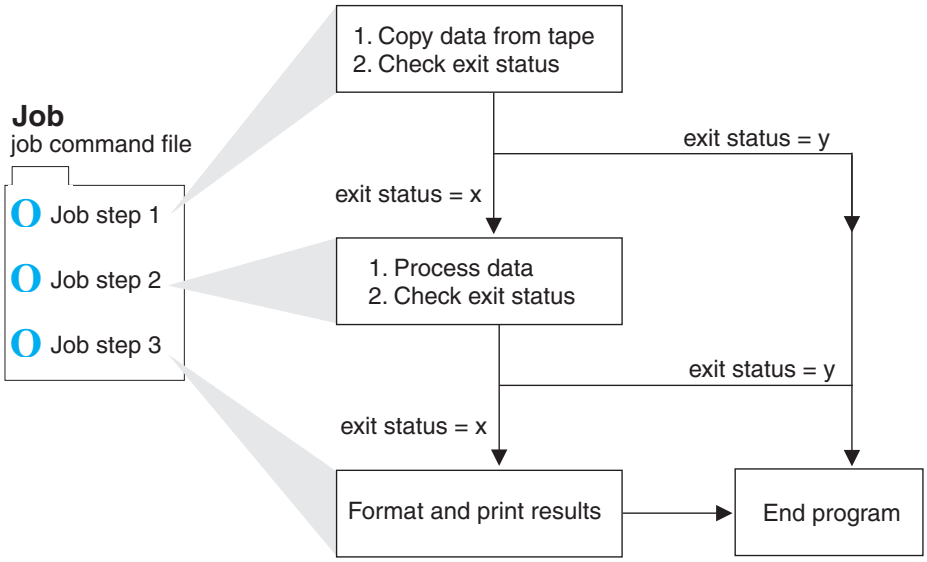

Figure 2. LoadLeveler job steps

Each of these job steps is defined in a single **job command file**. A job command file specifies the name of the job, as well as the job steps that you want to submit, and can contain other LoadLeveler statements.

LoadLeveler tries to execute each of your job steps on a machine that has enough resources to support executing and checkpointing each step. If your job command file has multiple job steps, the job steps will not necessarily run on the same machine, unless you explicitly request that they do.

You can submit batch jobs to LoadLeveler for scheduling. Batch jobs run in the background and generally do not require any input from the user. Batch jobs can either be **serial** or **parallel**. A serial job runs on a single machine. A parallel job is a program designed to execute as a number of individual, but related, processes on one or more of your system's nodes. When executed, these related processes can communicate with each other (through message passing or shared memory) to exchange data or synchronize their execution.

For parallel jobs, LoadLeveler interacts with Parallel Operating Environment (POE) to allocate nodes, assign tasks to nodes, and launch tasks.

Machine definition

For LoadLeveler to schedule a job on a machine, the machine must be a valid member of the LoadLeveler cluster. A cluster is the combination of all of the different types of machines that use LoadLeveler.

To make a machine a member of the LoadLeveler cluster, the administrator has to install the LoadLeveler software onto the machine and identify the central manager (described in "Roles of machines"). Once a machine becomes a valid member of the cluster, LoadLeveler can schedule jobs to it.

Roles of machines

Each machine in the LoadLeveler cluster performs one or more roles in scheduling jobs. Roles performed in scheduling jobs by each machine in the LoadLeveler cluster are as follows:

- **Job Manager Machine:** When a job is submitted, it gets placed in a queue managed by a job manager machine. This machine contacts another machine that serves as the central manager for the entire LoadLeveler cluster. This job manager machine asks the central manager to find a machine that can run the job, and also keeps persistent information about the job. Some job manager machines are known as *public job manager machines*, meaning that any LoadLeveler user can access them. These machines schedule jobs submitted from submit-only machines.
- **Central Manager Machine:** The role of the central manager is to examine the job's requirements and find one or more machines in the LoadLeveler cluster that will run the job. Once it finds the machine(s), it notifies the scheduling machine.
- **Executing Machine:** The machine that runs the job is known as the executing machine.
- **Resource Manager Machine:** The role of the resource manager is to collect status from all executing and job manager machines in the cluster.
- **Region Manager Machine:** The role of the region manager is to monitor node and adapter status by receiving heartbeats from the executing machines it manages.
- **Submitting Machine:** This type of machine is known as a *submit-only* machine. It participates in the LoadLeveler cluster on a limited basis. Although the name implies that users of these machines can only submit jobs, they can also query and cancel jobs. The submit-only machine feature allows workstations that are not part of the LoadLeveler cluster to submit jobs to the cluster.

Keep in mind that one machine can assume multiple roles, as shown in Figure 3 on page 7.

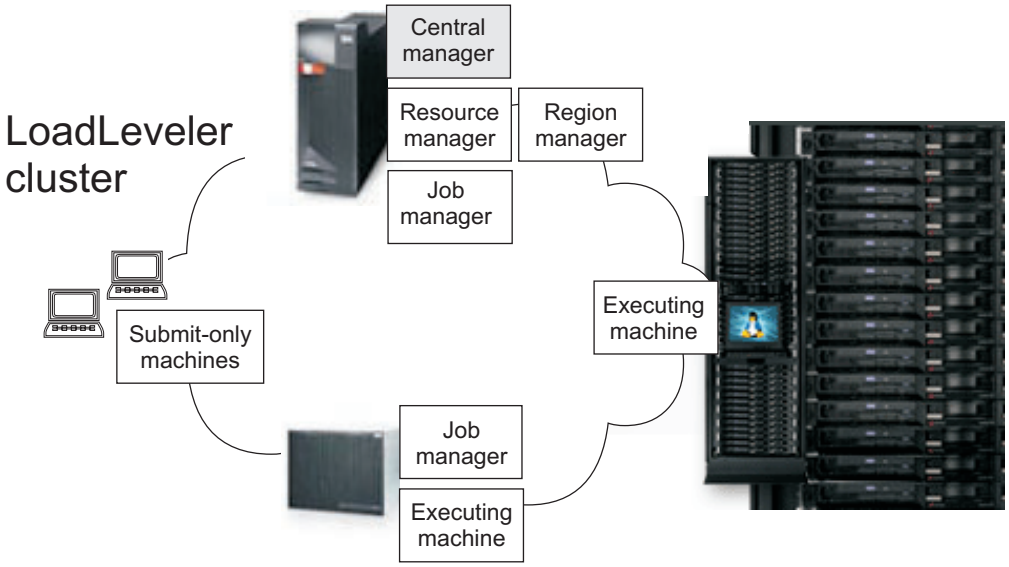

Figure 3. Multiple roles of machines

Machine availability

There may be times when some of the machines in the LoadLeveler cluster are not available to process jobs. For instance, when the owners of the machines have decided to make them unavailable. This ability of LoadLeveler to allow users to restrict the use of their machines provides flexibility and control over the resources.

Machine owners can make their personal workstations available to other LoadLeveler users in several ways. For example, you can specify that:

- The machine will always be available
- The machine will be available only between certain hours
- The machine will be available when the keyboard and mouse are not being used interactively.

Owners can also specify that their personal workstations never be made available to other LoadLeveler users.

How LoadLeveler schedules jobs

LoadLeveler consists of the following two components: the **resource manager**, which is responsible for managing all machine and job resources, and the **scheduler**, which is responsible for scheduling jobs on the resources provided by the resource manager.

When a user submits a job, LoadLeveler examines the job command file to determine what resources the job will need. LoadLeveler determines which machine, or group of machines, is best suited to provide these resources, then LoadLeveler dispatches the job to the appropriate machines. To aid this process, LoadLeveler uses queues.

A **job queue** is a list of jobs that are waiting to be processed. When a user submits a job to LoadLeveler, the job is entered into an internal database, which resides on one of the machines in the LoadLeveler cluster, until it is ready to be dispatched to run on another machine.

Once LoadLeveler examines a job to determine its required resources, the job is dispatched to a machine to be processed. A job can be dispatched to either one machine, or in the case of parallel jobs, to multiple machines. Once the job reaches the executing machine, the job runs.

Jobs do not necessarily get dispatched to machines in the cluster on a first-come, first-serve basis. Instead, LoadLeveler examines the requirements and characteristics of the job and the availability of machines, and then determines the best time for the job to be dispatched.

LoadLeveler also uses **job classes** to schedule jobs to run on machines. A job class is a classification to which a job can belong. For example, short running jobs may belong to a job class called `short_jobs`. Similarly, jobs that are only allowed to run on the weekends may belong to a class called `weekend`. The system administrator can define these job classes and select the users that are authorized to submit jobs of these classes.

You can specify which types of jobs will run on a machine by specifying the types of job classes the machine will support. LoadLeveler also examines a job's **priority** to determine when to schedule the job on a machine. A priority of a job is used to determine its position among a list of all jobs waiting to be dispatched.

"The LoadLeveler job cycle" on page 16 describes job flow in the LoadLeveler environment in more detail.

How LoadLeveler daemons process jobs

LoadLeveler has its own set of daemons that control the processes moving jobs through the LoadLeveler cluster.

The LoadLeveler daemons are programs that run continuously and control the processes that move jobs through the LoadLeveler cluster. A master daemon (**LoadL_master**) runs on all machines in the LoadLeveler cluster and manages other daemons.

Machine and adapter configuration and status changes are now detected by the region manager and the startd daemons.

Table 4 summarizes these daemons, which are described in further detail in topics immediately following the table.

Table 4. LoadLeveler daemons

Daemon	Description	Part of Resource Manager component	Part of Scheduler component
LoadL_master	Referred to as the master daemon. Runs on all machines in the LoadLeveler cluster and manages other daemons.	Yes	Yes
LoadL_schedd	Referred to as the Schedd daemon or the job manager. Receives jobs from the **llsubmit** command and manages them on machines selected by the negotiator daemon (as defined by the administrator).	Yes	No

Table 4. LoadLeveler daemons (continued)

Daemon	Description	Part of Resource Manager component	Part of Scheduler component
LoadL_startd	Referred to as the startd daemon. The startd daemon performs the following functions: • Monitors job and machine resources on local machines • Generates local adapter configuration • Forwards information to the negotiator, resource manager, and region manager daemons • Sends heartbeats to the region manager daemon The startd daemon spawns the starter process (**LoadL_starter**) which manages running jobs on the executing machine.	Yes	No
LoadL_region_mgr	Referred to as the region manager daemon. Detects and monitors machine and adapter status from the startd machines it manages and sends this information to the **LoadL_negotiator** and **LoadL_resource_mgr** daemons.	Yes	No
LoadL_resource_mgr	Referred to as the resource manager daemon. Collects all machine updates from the executing machine. Generates events to the scheduler for changes in the machine status. Maintains a list of all jobs managed by the resource manager. Responds to query requests for machine, job, and cluster information.	Yes	No
LoadL_kbdd	Referred to as the keyboard daemon. Monitors keyboard and mouse activity.	Yes	No
LoadL_negotiator	Referred to as the negotiator daemon or central manager. Monitors the status of each job and machine in the cluster. Responds to queries from **llstatus** and **llq** commands. Runs on the central manager machine.	No	Yes

The master daemon

The **master** daemon runs on every machine in the LoadLeveler cluster, except the submit-only machines. The real and effective user ID of this daemon must be **root**.

The **LoadL_master** binary is installed as a **setuid** program with the owner set to **root**. The **master** daemon and all daemons started by the **master** must be able to run with **root** privileges in order to switch the identity to the owner of any job being processed.

The **master** daemon determines whether to start any other daemons by checking the **START_DAEMONS** keyword in the global or local configuration file. If the keyword is set to **true**, the daemons are started. If the keyword is set to **false**, the **master** daemon terminates and generates a message.

The **master** daemon will not start on a Linux machine if **SEC_ENABLEMENT** is set to **CTSEC**. If the **master** daemon does not start, no other daemons will start.

On machines designated as primary or alternate for running the central manager, region manager, and resource manager, the **master** controls the failover function for these daemons. If a central manager, region manager, or resource manager machine failure is detected on a primary machine, the **master** will start the appropriate daemons on an alternate machine.

The **master** daemon starts and if necessary, restarts all of the LoadLeveler daemons that the machine it resides on is configured to run. This daemon also runs the **kbdd** daemon, which monitors keyboard and mouse activity.

When the **master** daemon detects a failure on one of the daemons that it is monitoring, it attempts to restart it. Because this daemon recognizes that certain situations may prevent a daemon from running, it limits its restart attempts to the number defined for the **RESTARTS_PER_HOUR** keyword in the configuration file. If this limit is exceeded, the **master** daemon forces all daemons including itself to exit.

When a daemon must be restarted, the **master** sends mail to the administrators identified by the **LOADL_ADMIN** keyword in the configuration file. The mail contains the name of the failing daemon, its termination status, and a section of the daemon's most recent log file. If the **master** aborts after exceeding **RESTARTS_PER_HOUR**, it will also send that mail before exiting.

The **master** daemon may perform the following actions in response to an **llctl** command:

- Kill all daemons and exit (**stop** keyword)
- Kill all daemons and execute a new **master** (**recycle** keyword)
- Reread the configuration files, stop or start daemons as appropriate for the new configuration files (**reconfig** keyword)
- Send drain request to startd and (**drain** keyword)
- Send flush request to startd and send result to caller (**flush** keyword)
- Send suspend request to startd and send result to caller (**suspend** keyword)
- Send resume request to startd and Schedd, and send result to caller (**resume** keyword)

The Schedd daemon

The **Schedd** or job manager daemon receives jobs sent by the **llsubmit** command and manages those jobs to machines selected by the negotiator daemon.

The **Schedd** daemon can be in any one of the following activity states:

Available
This machine is available to schedule jobs.

Drained
The **Schedd** machine accepts no more jobs. There are no jobs in starting or running state. Jobs in the Idle state are drained, meaning they will not get dispatched.

Draining
The **Schedd** daemon is being drained by the administrator but some jobs are still running. The state of the machine remains Draining until all running jobs complete. At that time, the machine status changes to Drained.

Down The daemon is not running on this machine. The **Schedd** daemon enters this state when it has not reported its status to the **negotiator**. This can occur when the machine is actually down, or because there is a network failure.

The **Schedd** daemon performs the following functions:

- Assigns new job identifiers when requested by the job submission process (for example, by the **llsubmit** command).
- Receives new jobs from the **llsubmit** command. A new job is received as a *job object* for each job step. A job object is the data structure in memory containing all the information about a job step. The **Schedd** forwards the job object to the **negotiator** daemon as soon as it is received from the submit command.
- Maintains on disk copies of jobs submitted locally (on this machine) that are either waiting or running on a remote (different) machine. The central manager can use this information to reconstruct the job information in the event of a failure. This information is also used for accounting purposes.
- Responds to directives sent by the administrator through the **negotiator** daemon. The directives include:
  - Run a job.
  - Change the priority of a job.
  - Remove a job.
  - Hold or release a job.
  - Send information about all jobs.
- Sends job events to the negotiator daemon when:
  - **Schedd** is restarting.
  - A new series of job objects are arriving.
  - A job is started.
  - A job was rejected, completed, removed, or vacated. **Schedd** determines the status by examining the exit status returned by the **startd**.
- Communicates with the Parallel Operating Environment (POE) when you run an interactive POE job.
- Requests that a remote **startd** daemon end a job.
- Receives accounting information from **startd**.
- Receives requests for reservations.
- Collects resource usage data when jobs terminate and stores it as historic fair share data in the $(SPOOL) directory.
- Sends historic fair share data to the central manager when it is updated or when the **Schedd** daemon is restarted.
- Maintains and stores records of historic CPU and IBM System Blue Gene Solution utilization for users and groups known to the **Schedd**.
- Passes the historic CPU and Blue Gene utilization data to the central manager.

The startd daemon

The **startd** daemon monitors the status of each job, reservation, and machine in the cluster, and forwards this information to the negotiator daemon. The **startd** daemon also receives and executes job requests originating from remote machines. The startd daemon generates its adapter information and sends it to the negotiator, resource, and region manager daemons. It also sends UDP heartbeat packets to the region manager daemon. The master daemon starts, restarts, signals, and stops the **startd** daemon.

The **startd** daemon can be in any one of the following states:

Busy The maximum number of jobs are running on this machine as specified by the **MAX_STARTERS** configuration keyword.

Down The daemon is not running on this machine. The **startd** daemon enters this state when it has not reported its status to the negotiator. This can occur when the machine is actually down, or because there is a network failure.

Drained
The **startd** machine will not accept any new jobs. No jobs are running when **startd** is in the drained state.

Draining
The **startd** daemon is being drained by the administrator, but some jobs are still running. The machine remains in the draining state until all of the running jobs have completed, at which time the machine status changes to drained. The **startd** daemon will not accept any new jobs while in the draining state.

Flush Any running jobs have been vacated (terminated and returned to the queue to be redispatched). The **startd** daemon will not accept any new jobs.

Idle The machine is not running any jobs.

None LoadLeveler is running on this machine, but no jobs can run here.

Running
The machine is running one or more jobs and is capable of running more.

Suspend
All LoadLeveler jobs running on this machine are stopped (cease processing), but remain in virtual memory. The **startd** daemon will not accept any new jobs.

The **startd** daemon performs these functions:

- Runs a time-out procedure that includes building a snapshot of the state of the machine that includes static and dynamic data. This time-out procedure is run at the following times:
  - After a job completes.
  - According to the definition of the **POLLING_FREQUENCY** keyword in the configuration file.
- Records the following information in LoadLeveler variables and sends the information to the negotiator.
  - State (of the startd daemon)
  - EnteredCurrentState
  - Memory
  - Disk
  - KeyboardIdle

  - Cpus
  - LoadAvg
  - Machine
  - Adapter
  - AvailableClasses
- Calculates the SUSPEND, RESUME, CONTINUE, and VACATE expressions through which you can manage job status.
- Receives job requests from the **Schedd** daemon to:
  - Start a job
  - Preempt or resume a job
  - Vacate a job
  - Cancel

  When the **Schedd** daemon tells the **startd** daemon to start a job, the **startd** determines whether its own state permits a new job to run as described in Table 5:

Table 5. startd determines whether its own state permits a new job to run

If:	Then this happens:
Yes, it can start a new job	The **startd** forks a **starter** process.
No, it cannot start a new job	The **startd** rejects the request for one of the following reasons: • Jobs have been suspended, flushed, or drained • The job limit set for the **MAX_STARTERS** keyword has been reached • There are not enough classes available for the designated job class

- Receives requests from the master (through the **llctl** command) to do one of the following:
  - Drain (**drain** keyword)
  - Flush (**flush** keyword)
  - Suspend (**suspend** keyword)
  - Resume (**resume** keyword)
- For each request, **startd** marks its own new state, forwards its new state to the negotiator daemon, and then performs the appropriate action for any jobs that are active.
- Receives notification of keyboard and mouse activity from the **kbdd** daemon
- Periodically examines the process table for LoadLeveler jobs and accumulates resources consumed by those jobs. This resource data is used to determine if a job has exceeded its job limit and for recording in the history file.
- Sends accounting information to Schedd.

The starter process

The **startd** daemon spawns a **starter** process after the **Schedd** daemon tells the **startd** daemon to start a job.

The starter process manages all the processes associated with a job step. The starter process is responsible for running the job and reporting status back to the **startd** daemon.

The starter process performs these functions:

- Processes the prolog and epilog programs as defined by the **JOB_PROLOG** and **JOB_EPILOG** keywords in the configuration. The job will not run if the prolog program exits with a return code other than zero.

- Handles authentication. This includes:
  - Authenticates AFS, if necessary
  - Verifies that the submitting user is *not* **root**
  - Verifies that the submitting user has access to the appropriate directories in the local file system.
- Runs the job by forking a child process that runs with the user ID and all groups of the submitting user. That child process creates a new process group of which it is the process group leader, and executes the user's program or a shell.

  The starter process is responsible for detecting the termination of any process that it forks. To ensure that all processes associated with a job are terminated after the process forked by the starter terminates, process tracking must be enabled. To configure LoadLeveler for process tracking, see "Tracking job processes" on page 73.
- Responds to vacate and suspend orders from the **startd**.

The region manager daemon

The region manager daemon detects and monitors node and adapter status from all of the startd machines it manages and sends the status information to the central and resource manager daemons.

The **region manager** daemon will only start up if a region is defined in the configuration. If the **region manager** daemon is not configured, the adapter and node status will only come from the configuration information available to the startd daemon and will not reflect the actual connectivity of the adapter or node.

The **region manager** daemon requires specific information, such as:

- The name of the primary machine on which it will run.
- The name of the backup or alternate machine.
- The region or list of machines reporting to it.

A LoadLeveler cluster can consist of more than one region and can have more than one region manager. A region, or managed region, consists of a list of machines that will send heartbeats and report information to the region manager.

The region manager node must have similar connectivity to the network as the startd nodes it manages, so that all of its configured network interfaces are able to connect to all of the startd node's network interfaces.

The **region manager** daemon performs the following functions:

- Maintains an in-memory database containing all machines in its managed region and their adapter ports
- Requests and receives adapter configuration information from the startds every time it starts up
- Updates its in-memory database whenever it receives adapter configuration information from a startd (for example, when a startd is reconfigured)
- Creates heartbeat IP pairs between the startd (source) and the region manager (destination) ports and sends them back to the corresponding startd
- Receives heartbeats from the startds
- Determines heartbeat status based on the UDP heartbeat packets from the startd daemon
- Sends heartbeat status to the central manager and resource manager

The resource manager daemon

The **resource manager** daemon collects all machine updates from the executing machine, generates events to the scheduler for changes in machine status, maintains a list of all jobs managed by the resource manager, and responds to query requests for machine, job, and cluster information.

The kbdd daemon

The **kbdd** daemon monitors keyboard and mouse activity. The **kbdd** daemon is spawned by the **master** daemon if the **X_RUNS_HERE** keyword in the configuration file is set to **true**.

The **kbdd** daemon notifies the **startd** daemon when it detects keyboard or mouse activity; however, **kbdd** is *not* interrupt driven. It sleeps for the number of seconds defined by the **POLLING_FREQUENCY** keyword in the LoadLeveler configuration file, and then determines if X events, in the form of mouse or keyboard activity, have occurred. For more information on the configuration file, see Chapter 5, "Defining LoadLeveler resources to administer," on page 89.

The negotiator daemon

The **negotiator** daemon maintains the status of each job and machine in the cluster and responds to queries from the **llstatus** and **llq** commands. The **negotiator** daemon runs on a single machine in the cluster (the central manager machine). This daemon is started, restarted, signalled, and stopped by the **master** daemon.

In a mixed cluster, the **negotiator** daemon must run on an AIX node.

The **negotiator** daemon receives status messages from each **Schedd** and **startd** daemons running in the cluster. The **negotiator** daemon tracks:

- Which **Schedd** daemons are running
- Which **startd** daemons are running, and the status of each **startd** machine.

If the **negotiator** does not receive an update from any machine within the time period defined by the **MACHINE_UPDATE_INTERVAL** keyword, then the **negotiator** assumes that the machine is down, and therefore the **Schedd** and **startd** daemons are also down.

The **negotiator** also maintains in its memory several queues and tables which determine where the job should run.

The **negotiator** performs the following functions:

- Receives and records job status changes from the **Schedd** daemon.
- Schedules jobs based on a variety of scheduling criteria and policy options. Once a job is selected, the **negotiator** contacts the **Schedd** that originally created the job.
- Handles requests to:
  - Set priorities
  - Query about jobs, machines, classes, and reservations
  - Change reservation attributes
  - Bind jobs to reservations
  - Remove a reservation
  - Remove a job
  - Hold or release a job
  - Favor or unfavor a user or a job.
- Receives notification of **Schedd** resets indicating that a **Schedd** has restarted.

The LoadLeveler job cycle

To illustrate the flow of job information through the LoadLeveler cluster, a description and sequence of diagrams have been provided.

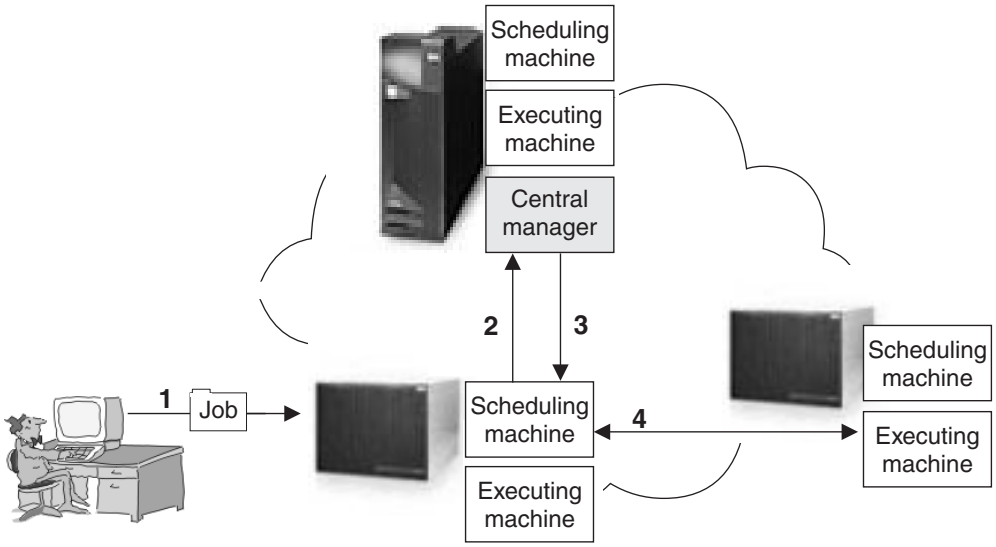

Figure 4. High-level job flow

The managing machine in a LoadLeveler cluster is known as the **central manager**. There are also machines that act as schedulers, and machines that serve as the executing machines. The arrows in Figure 4 illustrate the following:

- Arrow 1 indicates that a job has been submitted to LoadLeveler.
- Arrow 2 indicates that the scheduling machine contacts the central manager to inform it that a job has been submitted, and to find out if a machine exists that matches the job requirements.
- Arrow 3 indicates that the central manager checks to determine if a machine exists that is capable of running the job. Once a machine is found, the central manager informs the scheduling machine which machine is available.
- Arrow 4 indicates that the scheduling machine contacts the executing machine and provides it with information regarding the job. In this case, the scheduling and executing machines are different machines in the cluster, but they do not have to be different; the scheduling and executing machines may be the same physical machine.

Figure 4 is broken down into the following more detailed diagrams illustrating how LoadLeveler processes a job. The diagrams indicate specific job states for this example, but do not list all of the possible states for LoadLeveler jobs. A complete list of job states appears in "LoadLeveler job states" on page 19.

1. Submit a LoadLeveler job:

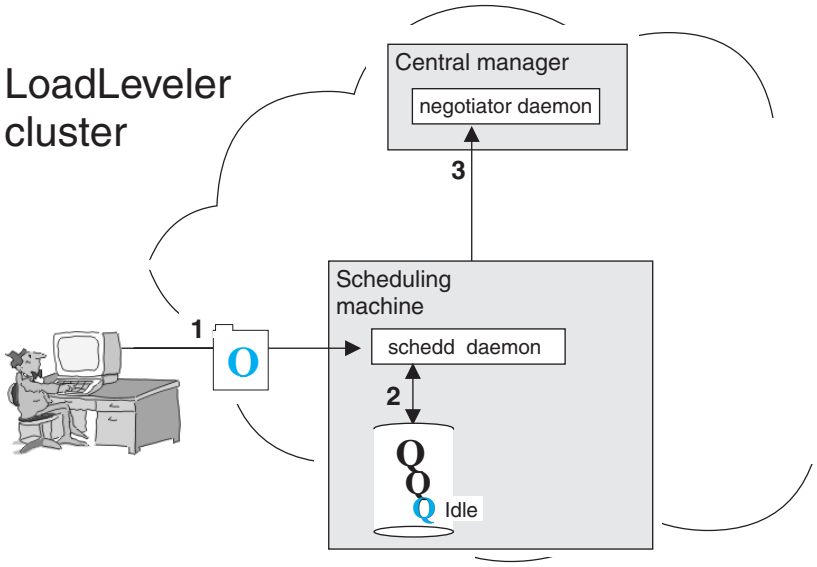

Figure 5. Job is submitted to LoadLeveler

Figure 5 illustrates that the Schedd daemon runs on the scheduling machine. This machine can also have the startd daemon running on it. The negotiator daemon resides on the central manager machine. The arrows in Figure 5 illustrate the following:

- Arrow 1 indicates that a job has been submitted to the scheduling machine.
- Arrow 2 indicates that the Schedd daemon, on the scheduling machine, stores all of the relevant job information on local disk.
- Arrow 3 indicates that the Schedd daemon sends job description information to the negotiator daemon. At this point, the submitted job is in the Idle state.

2. Permit to run:

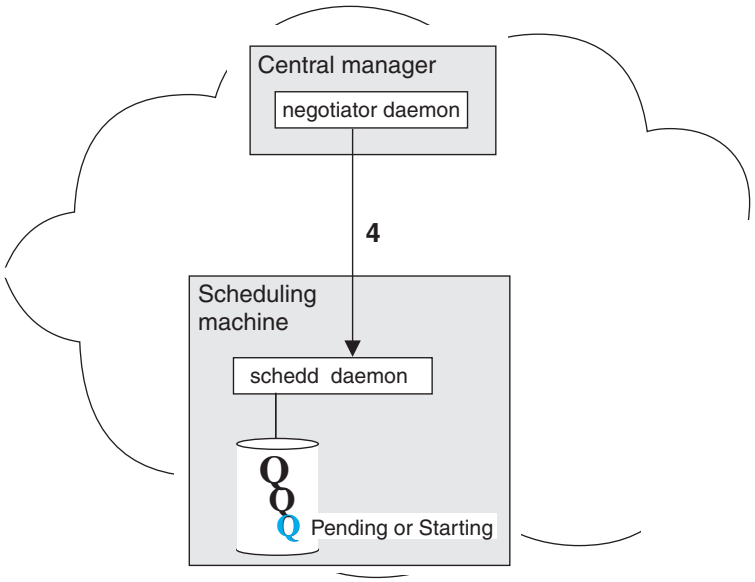

Figure 6. LoadLeveler authorizes the job

In Figure 6 on page 17, arrow 4 indicates that the negotiator daemon authorizes the Schedd daemon to begin taking steps to run the job. This authorization is called a *permit to run*. Once this is done, the job is considered Pending or Starting.

3. Prepare to run:

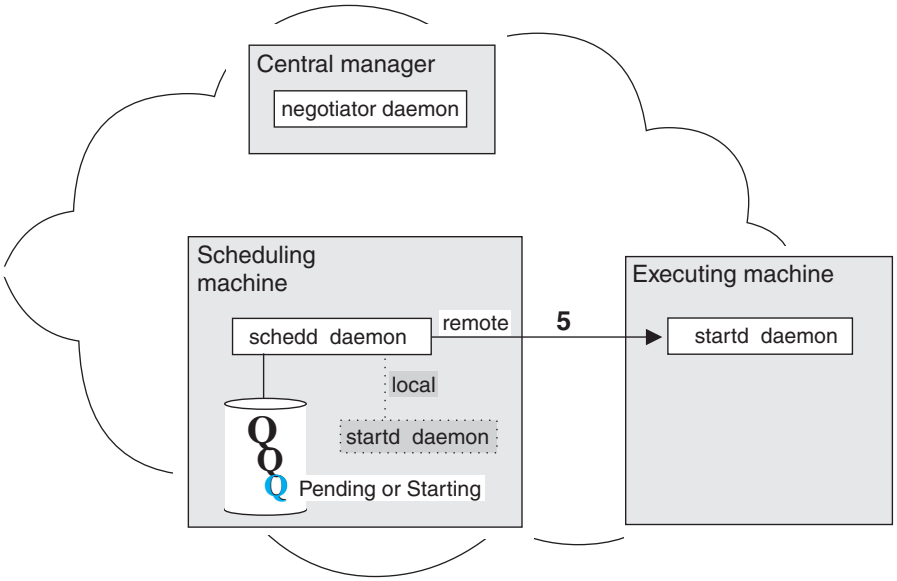

Figure 7. LoadLeveler prepares to run the job

In Figure 7, arrow 5 illustrates that the Schedd daemon contacts the startd daemon on the executing machine and requests that it start the job. The executing machine can either be a local machine (the machine to which the job was submitted) or another machine in the cluster. In this example, the local machine is **not** the executing machine.

4. Initiate job:

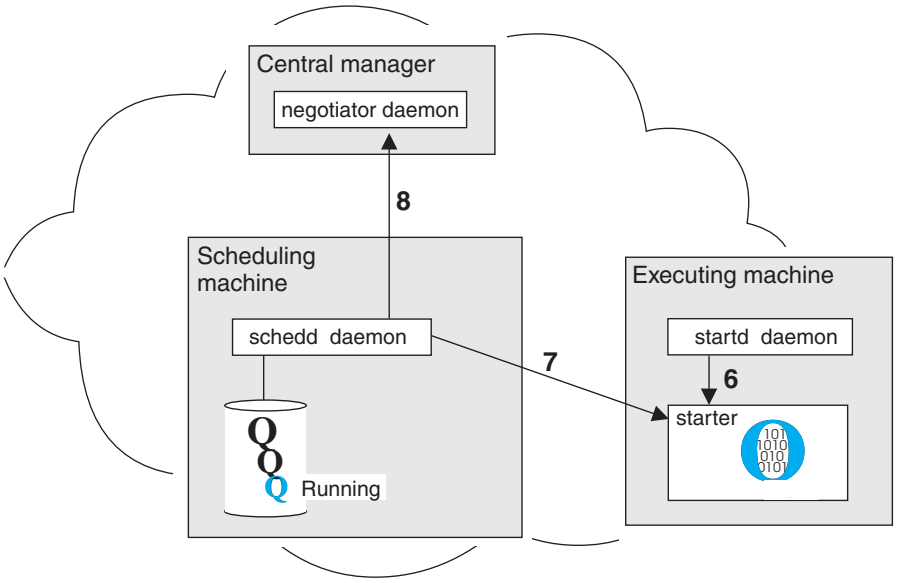

Figure 8. LoadLeveler starts the job

The arrows in Figure 8 on page 18 illustrate the following:

- Arrow 6 indicates that the **startd** daemon on the executing machine spawns a **starter** process for the job.
- Arrow 7 indicates that the Schedd daemon sends the starter process the job information and the executable.
- Arrow 8 indicates that the Schedd daemon notifies the negotiator daemon that the job has been started and the negotiator daemon marks the job as Running.

The starter forks and executes the user's job, and the starter parent waits for the child to complete.

5. Complete job:

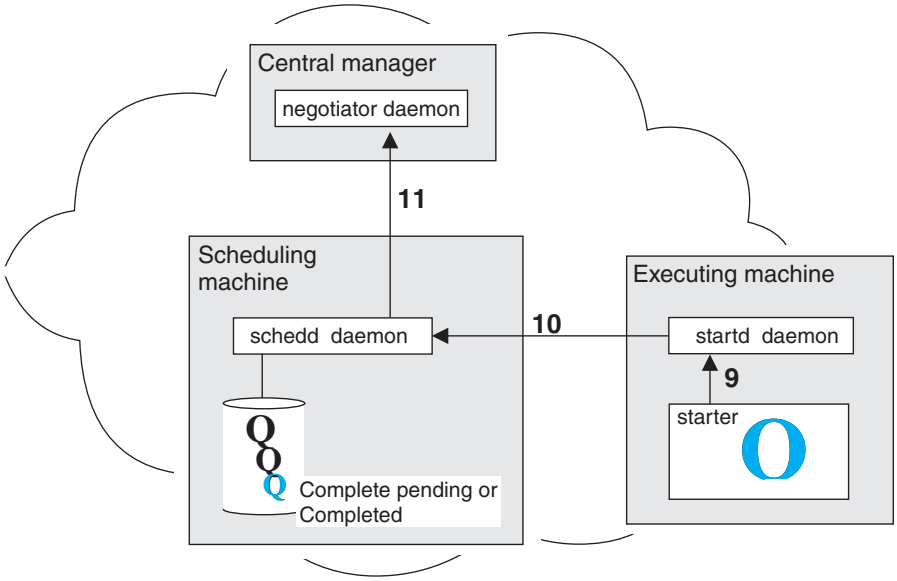

Figure 9. LoadLeveler completes the job

The arrows in Figure 9 illustrate the following:

- Arrow 9 indicates that when the job completes, the starter process notifies the startd daemon.
- Arrow 10 indicates that the startd daemon notifies the Schedd daemon.
- Arrow 11 indicates that the Schedd daemon examines the information it has received, and forwards it to the negotiator daemon. At this point, the job is in Completed or Complete Pending state.

LoadLeveler job states

As LoadLeveler processes a job, the job moves through various states. These states are listed in Table 6 on page 20. Job states that include "Pending," such as Complete Pending and Vacate Pending, are intermediate, temporary states.

Some options on LoadLeveler interfaces are valid only for jobs in certain states. For example, the **llmodify** command has options that apply only to jobs that are in the Idle state, or in states that are similar to it. To determine which job states are similar to the Idle state, use the "Similar to..." column in Table 6 on page 20, which indicates whether a particular job state is similar to the Idle, Running, or Terminating state. A dash (—) indicates that the state is not similar to an Idle, Running, or Terminating state.

Table 6. Job state descriptions and abbreviations

Job state	Similar to Idle or Running state?	Abbreviation in displays / output	Description
Canceled	Terminating	CA	The job was canceled either by a user or by an administrator.
Checkpointing	Running	CK	Indicates that a checkpoint has been initiated.
Completed	Terminating	C	The job has completed.
Complete Pending	Terminating	CP	The job is in the process of being completed.
Deferred	Idle	D	The job will not be assigned to a machine until a specified date. This date may have been specified by the user in the job command file, or may have been generated by the negotiator because a parallel job did not accumulate enough machines to run the job. Only the central manager places a job in the Deferred state.
Idle	Idle	I	The job is being considered to run on a machine, though no machine has been selected.
Not Queued	Idle	NQ	The job is not being considered to run on a machine. A job can enter this state because the associated Schedd is down, the user or group associated with the job is at its maximum **maxqueued** or **maxidle** value, or because the job has a dependency which cannot be determined. For more information on these keywords, see "Controlling the mix of idle and running jobs" on page 413. (Only the central manager places a job in the NotQueued state.)
Not Run	—	NR	The job will never be run because a dependency associated with the job was found to be false.
Pending	Running	P	The job is in the process of starting on one or more machines. (The negotiator indicates this state until the Schedd acknowledges that it has received the request to start the job. Then the negotiator changes the state of the job to Starting. The Schedd indicates the Pending state until all startd machines have acknowledged receipt of the start request. The Schedd then changes the state of the job to Starting.)
Preempted	Running	E	The job is preempted. This state applies only when LoadLeveler uses the suspend method to preempt the job.

Table 6. Job state descriptions and abbreviations (continued)

Job state	Similar to Idle or Running state?	Abbreviation in displays / output	Description
Preempt Pending	Running	EP	The job is in the process of being preempted. This state applies only when LoadLeveler uses the suspend method to preempt the job.
Rejected	Idle	X	The job is rejected.
Reject Pending	Idle	XP	The job did not start. Possible reasons why a job is rejected are: job requirements were not met on the target machine, or the user ID of the person running the job is not valid on the target machine. After a job leaves the Reject Pending state, it is moved into one of the following states: Idle, User Hold, or Removed.
Removed	Terminating	RM	The job was stopped by LoadLeveler.
Remove Pending	Terminating	RP	The job is in the process of being removed, but not all associated machines have acknowledged the removal of the job.
Resume Pending	Running	MP	The job is in the process of being resumed.
Running	Running	R	The job is running: the job was dispatched and has started on the designated machine.
Starting	Running	ST	The job is starting: the job was dispatched, was received by the target machine, and LoadLeveler is setting up the environment in which to run the job. For a parallel job, LoadLeveler sets up the environment on all required nodes. See the description of the “Pending” state for more information on when the negotiator or the Schedd daemon moves a job into the Starting state.
System Hold	Idle	S	The job has been put in system hold.

Table 6. Job state descriptions and abbreviations (continued)

Job state	Similar to Idle or Running state?	Abbreviation in displays / output	Description
Terminated	Terminating	TX	If the negotiator and Schedd daemons experience communication problems, they may be temporarily unable to exchange information concerning the status of jobs in the system. During this period of time, some of the jobs may actually complete and therefore be removed from the Schedd's list of active jobs. When communication resumes between the two daemons, the negotiator will move such jobs to the Terminated state, where they will remain for a set period of time (specified by the NEGOTIATOR_REMOVE_COMPLETED keyword in the configuration file). When this time has passed, the negotiator will remove the jobs from its active list.
User & System Hold	Idle	HS	The job has been put in both system hold and user hold.
User Hold	Idle	H	The job has been put in user hold.
Vacated	Idle	V	The job started but did not complete. The negotiator will reschedule the job (provided the job is allowed to be rescheduled). Possible reasons why a job moves to the Vacated state are: the machine where the job was running was flushed, the VACATE expression in the configuration file evaluated to True, or LoadLeveler detected a condition indicating the job needed to be vacated. For more information on the VACATE expression, see "Managing job status through control expressions" on page 72.
Vacate Pending	Idle	VP	The job is in the process of being vacated.

Consumable resources

Consumable resources are assets available on machines in your LoadLeveler cluster.

These assets are called "resources" because they model the commodities or services available on machines (including CPUs, real memory, virtual memory, large page memory, software licenses, disk space). They are considered "consumable" because job steps use specified amounts of these commodities when the step is running. Once the step finishes, the resource becomes available for another job step.

Consumable resources which model the characteristics of a specific machine (such as the number of CPUs or the number of specific software licenses available only on that machine) are called machine resources. Consumable resources which model resources that are available across the LoadLeveler cluster (such as floating software licenses) are called floating resources. For example, consider a

configuration with 10 licenses for a given program (which can be used on any machine in the cluster). If these licenses are defined as floating resources, all 10 can be used on one machine, or they can be spread across as many as 10 different machines.

The LoadLeveler administrator can specify:
- Consumable resources to be considered by LoadLeveler's scheduling algorithms
- Quantity of resources available on specific machines
- Quantity of floating resources available on machines in the cluster
- Consumable resources to be considered in determining the priority of executing machines
- Default amount of resources consumed by a job step of a specified job class
- Whether CPU, real memory, virtual memory, or large page resources should be enforced using Workload Manager (WLM)
- Whether all jobs submitted need to specify resources

Users submitting jobs can specify the resources consumed by each task of a job step, or the resources consumed by the job on each machine where it runs, regardless of the number of tasks assigned to that machine.

If affinity scheduling support is enabled, the CPUs requested in the consumable resources requirement of a job will be used by the scheduler to determine the number of CPUs to be allocated and attached to that job's tasks running on machines enabled for affinity scheduling. However, if the affinity scheduling request contains the processor-core affinity option, the number of CPUs will be determined from the value specified by the **task_affinity** keyword instead of the CPU's value in the consumable resources requirement. For more information on scheduling affinity, see "LoadLeveler scheduling affinity support" on page 147.

Notes:

1. When software licenses are used as a consumable resource, LoadLeveler does not attempt to obtain software licenses or to verify that software licenses have been obtained. However, by providing a user exit that can be invoked as a submit filter, the LoadLeveler administrator can provide code to first obtain the required license and then allow the job step to run. For more information on filtering job scripts, see "Filtering a job script" on page 79.
2. LoadLeveler scheduling algorithms use the availability of requested consumable resources to determine the machine or machines on which a job will run. Consumable resources (except for CPU, real memory, virtual memory and large page) are only used for scheduling purposes and are not enforced. Instead, LoadLeveler's negotiator daemon keeps track of the consumable resources available by reducing them by the amount requested when a job step is scheduled, and increasing them when a consuming job step completes.
3. If a job is preempted, the job continues to use all consumable resources except for ConsumableCpus and ConsumableMemory (real memory) which are made available to other jobs.

Consumable resources and Workload Manager

If the administrator has indicated that resources should be enforced, LoadLeveler uses Workload Manager (WLM) to give greater control over CPU, real memory, virtual memory and large page resource allocation.

WLM monitors system resources and regulates their allocation to processes. These actions prevent jobs from interfering with each other when they have conflicting

resource requirements. WLM achieves this control by creating different classes of service and allowing attributes to be specified for those classes.

LoadLeveler dynamically generates WLM classes with specific resource entitlements. A single WLM class is created for each job step and the process id of that job step is assigned to that class. This is done for each node that a job step is assigned to run on. LoadLeveler then defines resource shares or limits for that class depending on the LoadLeveler enforcement policy defined. These resource shares or limits represent the job's requested resource usage in relation to the amount of resources available on the machine.

When LoadLeveler defines multiple memory resources under one WLM class, WLM uses the following order to determine if resource limits have been exceeded:

1. Real Memory Absolute Limit
2. Virtual Memory Absolute Limit
3. Large Page Limit
4. Real Memory shares or percent limit

Note: When real memory or CPU with either shares or percent limits are exceeded, the job processes in that class receive a lower scheduling priority until their utilization drops below the hard max limit. When virtual memory or absolute real memory limits are exceeded, the job processes are killed. When the large page limit is exceeded, any new large page requests are denied.

When the enforcement policy is shares, LoadLeveler assigns a share value to the class based on the resources requested for the job step (one unit of resource equals one share). When the job step process is running, WLM dynamically calculates an appropriate resource entitlement based on the WLM class share value of the job step and the total number of shares requested by all active WLM classes. It is important to note that WLM will only enforce these target percentages when the resource is under contention.

When the enforcement policy is limits (soft or hard), LoadLeveler assigns a percentage value to the class based on the resources requested for the job step and the total machine resources. This resource percentage is enforced regardless of any other active WLM classes. A soft limit indicates the maximum amount of the resource that can be made available when there is contention for the resources. This maximum can be exceeded if no one else requires the resource. A hard limit indicates the maximum amount of the resource that can be made available even if there is no contention for the resources.

Note: A WLM class is active for the duration of a job step and is deleted when the job step completes. On AIX, there is a limit of 8192 active WLM classes per machine. Therefore, when resources are being enforced on AIX, only 8192 job steps can be running on one machine.

For additional information about integrating LoadLeveler with Workload Manager, see “Steps for integrating LoadLeveler with the Workload Manager” on page 133.

Overview of reservations

Under the BACKFILL scheduler only, LoadLeveler allows authorized users to make reservations, which specify a time period during which specific node resources are reserved for exclusive use by particular users or groups. This capability is known in the computing industry as advance reservation.

Normally, jobs wait to be dispatched until the resources they require become available. Through the use of reservations, wait time can be reduced because the jobs have exclusive use of the node resources (CPUs, memory, disk drives, communication adapters, and so on) as soon as the reservation period begins.

Note: Advance reservation supports Blue Gene resources including the Blue Gene compute nodes. For more information, see "Blue Gene reservation support" on page 157.

In addition to reducing wait time, reservations also are useful for:

- Running a workload that needs to start or finish at a particular time. The job steps must be associated with, or bound to, the reservation before LoadLeveler can run them during the reservation period.
- Running a workload that needs to start as soon as possible on a set of machines.
- Reserving resources for a workload that repeats at regular intervals. You can make a single request to create a recurring reservation, which reserves a specific set of resources for a specific time slot that repeats on a regular basis for a defined interval.
- Setting aside a set of nodes for maintenance purposes.

Only bound job steps may run on the reserved nodes, which means that a bound job step competes for reserved resources only with other job steps that are bound to the same reservation.

There are different types of reservations:

One-time reservation
: A reservation with a specified start time and duration.

Recurring reservation
: A reservation with a specified start time and duration that reoccurs for a specified period of time.

Flexible reservation
: A reservation with a specified duration that will start as soon as the resources it requests become available. A flexible reservation cannot be made to recur.

The following sequence of events describes, in general terms, how you can set up and use reservations in the LoadLeveler environment. It also describes how LoadLeveler manages activities related to the use of reservations.

1. **Configuring LoadLeveler to support reservations**

   An administrator uses specific keywords in the configuration and administration files to define general reservation policies. These keywords include:

   - **max_reservations**, when used in the *global configuration* file defines the maximum number of reservations for the entire cluster.
   - **max_reservations**, when used in a user or group stanza of the *administration* file can also be used to define both:
     - The users or groups that will be allowed to create reservations. To be authorized to create reservations, LoadLeveler administrators also must have the **max_reservations** keyword set in their own user or group stanzas.
     - How many reservations users may own.

Note: With recurring advance reservations, to avoid confusion about what counts as one reservation, LoadLeveler is using the approach that one reservation counts as one instance regardless of the number of times the reservation recurs before it expires. This applies to the system wide **max_reservations** configuration setting as well as the same type of configuration settings at the user and group levels.

- **max_reservation_duration**, which defines the maximum duration for reservations.
- **reservation_permitted**, which defines the nodes that may be used for reservations.
- **max_reservation_expiration** which defines how long recurring reservations are permitted to last (expressed as the number of days).
- **reservation_type**, which specifies what types of reservations can be used by a user or a group. The types are **ALL**, **FLEXIBLE**, and **NONE**.

Administrators also may configure LoadLeveler to collect accounting data about reservations when the reservations complete or are canceled.

2. **Creating reservations**

   After LoadLeveler is configured for reservations, an administrator or authorized user may create specific reservations, defining reservation attributes that include:
   - The start time and the duration of the reservation. The start and end times for a reservation are based on the time-of-day (TOD) clock on the central manager machine.
   - Whether the reservation is a flexible reservation. Flexible reservations do not specify a start time.
   - Whether or not the reservation recurs and if it recurs, the interval in which it does so.
   - The nodes to be reserved. Until the reservation period actually begins, the selected nodes are available to run any jobs; when the reservation starts, only jobs bound to the reservation may run on the reserved nodes.
   - The users or groups that may use the reservation.
   - A user-supplied program to be invoked whenever the reservation state changes.
   - Floating resources for the reservation.
   - For a flexible reservation, whether it can contain nodes that are down.

   LoadLeveler assigns a unique ID to the reservation, and returns that ID to the owner.

   After the reservation is successfully created:

   - Reservation owners may:
     - Modify, query, and cancel their reservations.
     - Allow other LoadLeveler users or groups to submit jobs to run during a reservation period.
     - Submit jobs to run during a reservation period.
   - Users or groups that are allowed to use the reservation also may query reservations, and submit jobs to run during a reservation period. To run jobs during a reservation period, users must bind job steps to the reservation. You may bind both batch and interactive POE job steps to a reservation.

3. **Preparing for the start of a reservation**

   During the preparation time for a reservation, LoadLeveler:
   - Preempts any jobs that are still running on the reserved nodes.

- Checks the condition of reserved nodes, and notifies the reservation owner and LoadLeveler administrators by e-mail of any situations that might require the reservation owner or an administrator to take corrective action. Such conditions include:
  - Reserved nodes that are down, suspended, no longer in the LoadLeveler cluster, or otherwise unavailable for use.
  - Non-preemptable job steps that cannot finish running before the reservation start time.

  During this time, reservation owners may modify, cancel, and add users or groups to their reservations. Owners and users or groups that are allowed to use the reservation may query the reservation or bind job steps to it.

4. **Starting the reservation**

   When the reservation period begins, LoadLeveler dispatches job steps that are bound to the reservation.

   After the reservation period begins, reservation owners may modify, cancel, and add users or groups to their reservations. Owners and users or groups that are allowed to use the reservation may query the reservation or bind job steps to it.

   During the reservation period, LoadLeveler ignores system preemption rules for bound job steps; however, LoadLeveler administrators may use the **llpreempt** command to manually preempt bound job steps.

When the reservation ends or is canceled:
- LoadLeveler unbinds all job steps from the reservation if there are no further occurrences remaining. At this point the unbound job steps compete with all other LoadLeveler jobs for available resources. If there are occurrences remaining in the reservation, job steps are automatically bound to the next occurrence.
- For flexible reservations, all running and idle jobs will be canceled when the reservation ends.
- If accounting data is being collected for the reservation, LoadLeveler also updates the reservation history file.

For more detailed information and instructions for setting up and using reservations, see:
- "Configuring LoadLeveler to support reservations" on page 127.
- "Working with reservations" on page 203.

Fair share scheduling overview

Fair share scheduling in LoadLeveler provides a way to divide resources in a LoadLeveler cluster among users or groups of users.

Historic resource usage data that is collected at the time the job ends can be used to influence job priorities to achieve the resource usage proportions allocated to users or groups of users in the LoadLeveler configuration files. The resource usage data will decay over time so that the relatively recent historic resource usage will have the most influence on job priorities. The CPU resources in the cluster and the Blue Gene resources are currently supported by fair share scheduling.

For information about configuring fair share scheduling in LoadLeveler, see "Using fair share scheduling" on page 158.

Chapter 2. Getting a quick start using the default configuration

If you are very familiar with UNIX and Linux system administration, and job scheduling, follow the steps in this section to get LoadLeveler up and running on your network quickly in a default configuration.

This default configuration will merely enable you to submit serial jobs; for a more complex setup, see Chapter 4, "Configuring the LoadLeveler environment," on page 39.

What you need to know before you begin

LoadLeveler sets up default values for configuration information.

- **loadl** is the recommended LoadLeveler user ID and the LoadLeveler group ID. LoadLeveler daemons run under this user ID to perform file I/O, and many LoadLeveler files are owned by this user ID.
- The home directory of **loadl** is the configuration directory.
- **LoadL_config** is the name of the configuration file.

For information about configuration file keyword syntax and other details, see Chapter 10, "Configuration keyword reference," on page 231.

Using the default configuration files

Follow these steps to use the default configuration files.

Note: You can find samples of the **LoadL_admin** and **LoadL_config** files in the **release** directory (in the **samples** subdirectory).

1. Ensure that the installation procedure has completed successfully and that the configuration file, **LoadL_config**, exists in LoadLeveler's home directory or in the directory specified by the **LoadLConfig** keyword.
2. Identify yourself as the LoadLeveler administrator in the **LoadL_config** file using the **LOADL_ADMIN** keyword. The syntax of this keyword is:

   LOADL_ADMIN = *list_of_user_names* **(required)**

   Where *list_of_user_names* is a blank-delimited list of those individuals who will have administrative authority.

   Refer to "Defining LoadLeveler administrators" on page 45 for more information.
3. Define a machine to act as the LoadLeveler central manager in one of the following ways:
   - Code a machine stanza as follows in the administration file, which is called **LoadL_admin**. (Replace *machine_name* with the actual name of the machine.)

     ```
     machine_name: type = machine

     central_manager = true
     ```

     Note: Do not specify more than one machine as the central manager. Also, if during installation, you ran **llinit** with the **-cm** flag, the central manager is already defined in the **LoadL_admin** file because the **llinit** command takes

parameters that you entered and updates the administration and configuration files. See "Defining machines" on page 89 for more information.

- Alternatively, define one machine to act as the *primary* central manager and one or more machines to act as the *alternate* LoadLeveler central managers by specifying the following in the **LoadL_config** configuration file:

```
CENTRAL_MANAGER_LIST = primary_central_manager_machine \
                       [alternate_central_manager_machine_list]
```

Note: Because you cannot specify more than one machine as the primary central manager, the first machine name specified after the `CENTRAL_MANAGER_LIST` keyword in the configuration file automatically overrides the `central_manager` statement in the machine stanza in the administration file.

4. Define one machine to act as the *primary* resource manager and one or more machines to act as the *alternate* LoadLeveler resource managers by specifying the following in the **LoadL_config** configuration file:

```
RESOURCE_MGR_LIST = primary_resource_manager_machine \
                    [alternate_resource_manager_machine_list]
```

LoadLeveler for Linux quick start

If you would like to quickly install and configure LoadLeveler for Linux and submit a serial job on a single node, use these procedures.

Note: This setup is for a single node only and the node used for this example is: **c197blade1b05.ppd.pok.ibm.com**.

Quick installation

Details of this installation apply to RHEL 6 System x servers.

Note: This installation method is, however, applicable to all other systems. You must install the corresponding license RPM for the system you are installing on. This installation assumes that the LoadLeveler RPMs are located at: **/mnt/cdrom/**.

1. Log on to node `c197blade1b05.ppd.pok.ibm.com` as root, which is the node you are installing on.
2. Add a UNIX group for LoadLeveler users (make sure the group ID is correct) by entering the following command:

```
groupadd -g 1000 loadl
```

3. Add a UNIX user for LoadLeveler (make sure the user ID is correct) by entering the following command:

```
useradd -c "LoadLeveler User" -d /home/loadl -s /bin/bash -u 1001 -g 1000 -m loadl
```

4. Install the license RPM by entering the following command:

```
rpm -ivh /mnt/cdrom/LoadL-full-license-<OS-ARCH>-<install_version>.<arch>.rpm
```

5. Change to the LoadLeveler installation path by entering the following the command:

```
cd /opt/ibmll/LoadL/sbin
```

6. Run the LoadLeveler installation script by entering:

```
./install_ll -y -d /mnt/cdrom
```

7. Install the required LoadLeveler services updates for this RPM.

Updates and installation instructions are available at Fix Central (http://www-933.ibm.com/support/fixcentral/?productGroup0=ibm/fcpower&productGroup1=ibm/ClusterSoftware&productGroup2=ibm/power/IBM).

Quick configuration

Use this method to perform a quick configuration.

1. Change the log in to the newly created LoadLeveler user by entering the following command:

   ```
   su - loadl
   ```

2. Run the LoadLeveler initialization script:

   ```
   /opt/ibmll/LoadL/full/bin/llinit -local /tmp/loadl -release /opt/ibmll/LoadL/full -cm
   c197blade1b05.ppd.pok.ibm.com
   ```

Quick verification

Use this method to perform a quick verification.

1. Start LoadLeveler by entering the following command:

   ```
   llctl start
   ```

   You should receive a response similar to the following:

   ```
   llctl: Attempting to start LoadLeveler on host c197blade1b05.ppd.pok.ibm.com
   LoadL_master 5.1.0.0 rsnepXXXXa 2011/XX/XX RHEL 6.0 220
   CentralManager = c197blade1b05.ppd.pok.ibm.com
   [loadl@c197blade1b05 bin]$
   ```

2. Check LoadLeveler status by entering the following command:

   ```
   llstatus
   ```

   You should receive a response similar to the following:

   ```
   Active 2/4
   Schedd 2/4 0 job steps
   Startd 2/4 0 running tasks

   The Central Manager is defined on c197blade4b10.ppd.pok.ibm.com

   Absent: 2
   Startd: Down Drained Draining Flush Suspend
           0    0       0        0     0
   Schedd: Down Drained Draining
           0    0       0
   ```

   View LoadLeveler status by machine level, by entering the following command:

   ```
   llstatus -L machine
   ```

   Or alternatively, set the environment variable **LOADL_STATUS_LEVEL=machine** and issue:

   ```
   llstatus
   ```

   You should receive a response similar to the following:

   ```
   Name                      Schedd InQ  Act Startd Run LdAvg Idle Arch      OpSys
   c197blade4b10.ppd.pok.ibm Avail   0    0  Idle     0 0.00  9999 ppc64     Linux2
   c197blade4b11.ppd.pok.ibm Avail   0    0  Idle     0 0.00  9999 ppc64     Linux2

   ppc64/Linux2             2 machines      0  jobs      0  running tasks
   Total Machines           2 machines      0  jobs      0  running tasks

   The Central Manager is defined on c197blade4b10.ppd.pok.ibm.com

   The BACKFILL scheduler is in use

   The following 2 machines are marked absent
   c197blade4b09.ppd.pok.ibm.com
   c197blade4b12.ppd.pok.ibm.com
   ```

3. Submit a sample job, by entering the following command:

   ```
   llsubmit /opt/ibmll/LoadL/full/samples/job1.cmd
   ```

   You should receive a response similar to the following:

   ```
   llsubmit: The job "c197blade4b10.ppd.pok.ibm.com.1" with 2 job steps has been submitted.
   ```

4. Display the LoadLeveler job queue, by entering the following command:

   ```
   llq
   ```

   You should receive a response similar to the following:

   ```
   Id                       Owner      Submitted   ST PRI Class        Running On
   ------------------------ ---------- ----------- -- --- ------------ -----------
   c197blade4b10.1.0       loadl       5/15 17:25 R  50  No_Class     c197blade4b10
   c197blade4b10.1.1       loadl       5/15 17:25 R  50  No_Class     c197blade4b10
   2 job step(s) in queue, 0 waiting, 0 pending, 2 running, 0 held, 0 preempted
   ```

5. Check output files into the home directory (**/home/loadl**) by entering the following command:

   ```
   ls -ltr job*
   ```

   You should receive a response similar to the following:

   ```
   -rw-r--r--  1 loadl loadl 1487 May 15 17:26 job1.c197blade4b10.1.0.out
   -rw-r--r--  1 loadl loadl 1487 May 15 17:27 job1.c197blade4b10.1.1.out
   ```

Post-installation considerations

This information explains how to start (or restart) and stop LoadLeveler. It also tells you where files are located after you install LoadLeveler, and it points you to troubleshooting information.

Starting LoadLeveler

You can start LoadLeveler using any LoadLeveler administrator user ID as defined in the configuration file.

To start all of the machines that are defined in machine stanzas in the administration file, enter:

```
llctl -g start
```

The central manager machine is the first started, followed by other machines in the order listed in the administration file. See the **llctl** command in *LoadLeveler: Command and API Reference* for more information.

By default, **llctl** uses **rsh** to start LoadLeveler on the target machine. Other mechanisms, such as **ssh** can be used by setting the **LL_RSH_COMMAND** configuration keyword in **LoadL_config**. However you choose to start LoadLeveler on remote hosts, you must have the authority to run commands remotely on that host.

You can verify that the machine has been properly configured by running the sample jobs in the appropriate **samples** directory (job1.cmd, job2.cmd, and job3.cmd). You must read the job2.cmd and job3.cmd files before submitting them because job2 must be edited and a C program must be compiled to use job3. It is a good idea to copy the sample jobs to another directory before modifying them; you must have read/write permission to the directory in which they are located. You can use the **llsubmit** command to submit the sample jobs from several different machines and verify that they complete (see the **llsubmit** command in *LoadLeveler: Command and API Reference*).

If you are running AFS and some jobs do not complete, you might need to use the AFS **fs** command (**fs listacl**) to ensure that the you have write permission to the **spool**, **execute**, and **log** directories.

If you are running with cluster security services enabled and some jobs do not complete, ensure that you have write permission to the **spool**, **execute**, and **log** directories. Also ensure that the user ID is authorized to run jobs on the submitting machine (the identity of the user must exist in the **.rhosts** file of the user on the machine on which the job is being run).

Note: LoadLeveler for Linux does not support cluster security services.

If you are running submit-only LoadLeveler, once the LoadLeveler cluster is up and running, you can use the **llsubmit, llq,** and **llcancel** commands from the submit-only machines. For more information about these commands, see *LoadLeveler: Command and API Reference*.

Directory considerations

After installation, the product directories reside on disk. The installation process creates only those directories required to service the LoadLeveler options specified during the installation. See the *LoadLeveler: Installation Guide* for more information on the location of directories.

If you have a mixed LoadLeveler cluster of AIX and Linux machines, you might want to make the following symbolic links:

- On AIX, as **root**, enter:

```
mkdir -p /opt/ibmll
ln -s /usr/lpp/LoadL /opt/ibmll/LoadL
```

- On Linux, as **root**, enter:

```
mkdir -p /usr/lpp
ln -s /opt/ibmll/LoadL /usr/lpp/LoadL
```

With the addition of these symbolic links, a user application can use either **/usr/lpp/LoadL** or **/opt/ibmll/LoadL** to refer to the location of LoadLeveler files regardless of whether the application is running on AIX or Linux.

If LoadLeveler will not start following installation, see “Why won't LoadLeveler start?” on page 392 for troubleshooting information.

Chapter 3. What operating systems are supported by LoadLeveler?

LoadLeveler supports three operating systems.

- **AIX 7.1**

  IBM's AIX 7.1 is an open UNIX operating environments that conform to The Open Group UNIX 98 Base Brand industry standard. AIX 7.1 provides high levels of integration, flexibility, and reliability and it operates on IBM Power® Systems and IBM Cluster 1600 servers and workstations.

  AIX 7.1 supports the concurrent operation of 32- and 64-bit applications, with key internet technologies such as Java and XML parser for Java included as part of the base operating system.

  A strong affinity between AIX and Linux permits popular applications developed on Linux to run on AIX 7.1 with a simple recompilation.

- **Linux**

  LoadLeveler supports the following distribution of Linux:

  – Red Hat Enterprise Linux (RHEL) 6.2 on LoadLeveler for Linux on POWER
  – SUSE Linux Enterprise Server (SLES) 11 on LoadLeveler for Linux on x86 Architecture
  – RHEL 6.2 on LoadLeveler for Linux on x86 Architecture

- **IBM System Blue Gene Solution**

  While no LoadLeveler processes actually run on the Blue Gene machine, LoadLeveler can interact with the Blue Gene machine and supports the scheduling of jobs to the machine.

LoadLeveler for AIX and LoadLeveler for Linux compatibility

LoadLeveler for Linux is compatible with LoadLeveler for AIX. Its command line interfaces and application programming interfaces (APIs) are the same as they have been for AIX. The formats of the job command file, configuration file, and administration file also remain the same.

System administrators can set up and maintain a LoadLeveler cluster consisting of some machines running LoadLeveler for AIX and some machines running LoadLeveler for Linux. This is called a mixed cluster. In this mixed cluster jobs can be submitted from either AIX or Linux machines. Jobs submitted to a Linux job queue can be dispatched to an AIX machine for execution, and jobs submitted to an AIX job queue can be dispatched to a Linux machine for execution.

Although the LoadLeveler products for AIX and Linux are compatible, they do have some differences in the level of support for specific features. For further details, see the following topics:

- “Restrictions for LoadLeveler for Linux” on page 36.
- “Features not supported in LoadLeveler for Linux” on page 36.
- “Restrictions for LoadLeveler for AIX and LoadLeveler for Linux mixed clusters” on page 36.

Restrictions for LoadLeveler for Linux

LoadLeveler for Linux supports a subset of the features that are available in the LoadLeveler for AIX product.

The following features are available, but are subject to restrictions:

- LoadLeveler resource manager for Linux supports the 64-bit LoadLeveler API library (libllrapi.so) on RHEL 6.2 and SLES 11 on IBM xSeries® servers with AMD Opteron or Intel EM64T processors
- Support for AFS file systems

  LoadLeveler for Linux support for authenticated access to AFS file systems is limited to RHEL 6.2 on IBM xSeries® servers with AMD Opteron or Intel EM64T processors.
- Support for Workload Management (WLM):
  - Hard policy for ConsumableCpus is not available for WLM for Linux.
  - Large page memory supports only a 16 MB page size.

Features not supported in LoadLeveler for Linux

LoadLeveler for Linux supports a subset of the features that are available in the LoadLeveler for AIX product.

The following features are not supported:

- CtSec security

  LoadLeveler for AIX can exploit CtSec (Cluster Security Services) security functions. These functions authenticate the identity of users and programs interacting with LoadLeveler. These features are not available in this release of LoadLeveler for Linux.
- System error log

  Each LoadLeveler daemon has its own log file where information relevant to its operation is recorded. In addition to this feature which exists on all platforms, LoadLeveler for AIX also uses the errlog function to record critical LoadLeveler events into the AIX system log. Support for an equivalent Linux function is not available in this release.

Restrictions for LoadLeveler for AIX and LoadLeveler for Linux mixed clusters

Several restrictions apply when operating a LoadLeveler cluster that contains AIX and Linux machines.

When operating a LoadLeveler cluster that contains AIX and Linux machines, the following restrictions apply:

- The central manager node must run a version of LoadLeveler equal to or higher than any LoadLeveler version being run on a node in the cluster.
- CtSec security features cannot be used.
- Jobs that use checkpointing must be sent to nodes with the same operating system for execution. This can be done by either defining and specifying job checkpointing for job classes that exist only on AIX or Linux nodes or by coding appropriate requirements expressions.

Part 2. Configuring and managing the LoadLeveler environment

After installing IBM LoadLeveler, you may customize it by modifying both the **configuration** file and the **administration** file.

The configuration file contains many parameters that you can set or modify that will control how LoadLeveler operates. The administration file optionally lists and defines the machines in the LoadLeveler cluster and the characteristics of classes, users, and groups.

To easily manage LoadLeveler, you should have one global configuration file and only one administration file, both centrally located on a machine in the LoadLeveler cluster. Every other machine in the cluster must be able to read the configuration and administration file that are located on the central machine.

You may have multiple local configuration files that specify information specific to individual machines.

LoadLeveler does not prevent you from having multiple copies of administration files, but you need to be sure to update all the copies whenever you make a change to one. Having only one administration file prevents any confusion.

LoadLeveler also supports having the configuration in a database.

Chapter 4. Configuring the LoadLeveler environment

One of your main tasks as system administrator is to configure LoadLeveler.

To configure LoadLeveler, you need to know the following configuration information:
- The LoadLeveler user ID and group ID
- The source of the configuration
- The location of the configuration

Configuring LoadLeveler involves modifying the configuration data that specify the terms under which LoadLeveler can use machines.

Table 7 identifies where you can find more information about using either the configuration and administration files or using a database as the source for LoadLeveler configuration, and keywords to modify the LoadLeveler environment.

Table 7. Roadmap of tasks for LoadLeveler administrators

To learn about:	Read the following:
Controlling how LoadLeveler operates by customizing the global and local configuration files or a database	Chapter 4, "Configuring the LoadLeveler environment"
Customizing LoadLeveler resources	Chapter 5, "Defining LoadLeveler resources to administer," on page 89
Additional ways to customize LoadLeveler configuration	Chapter 6, "Performing additional administrator tasks," on page 109

To control or monitor LoadLeveler operations by using the LoadLeveler commands and APIs, see *LoadLeveler: Command and API Reference*.

You can run your installation with default values set by LoadLeveler, or you can change any or all of them. Table 8 lists topics that discuss how you may configure the LoadLeveler environment by modifying the configuration file.

Table 8. Roadmap of administrator tasks related to using or modifying the LoadLeveler configuration

To learn about:	Read the following:
Using the default configuration files shipped with LoadLeveler	Chapter 2, "Getting a quick start using the default configuration," on page 29
Modifying LoadLeveler user and the source of the configuration	"The master configuration file" on page 40

Table 8. Roadmap of administrator tasks related to using or modifying the LoadLeveler configuration (continued)

To learn about:	Read the following:
Defining major elements of the LoadLeveler configuration	• "Defining LoadLeveler administrators" on page 45 • "Defining a LoadLeveler cluster" on page 45 • "Defining LoadLeveler machine characteristics" on page 59 • "Defining security mechanisms" on page 60 • "Defining usage policies for consumable resources" on page 65 • "Steps for configuring a LoadLeveler multicluster" on page 151
Enabling optional LoadLeveler functions	• "Gathering job accounting data" on page 65 • "Managing job status through control expressions" on page 72 • "Tracking job processes" on page 73 • "Querying multiple LoadLeveler clusters" on page 74
Modifying LoadLeveler operations through installation exits	"Providing additional job-processing controls through installation exits" on page 75

The master configuration file

By default, LoadLeveler uses the **loadl** user name and **loadl** group name to set the effective user ID and group ID to run LoadLeveler daemons. Also by default, LoadLeveler will use a set of configuration files as the source for configuration data. It expects that the global configuration file will be named **LoadL_config**and will be found in the **loadl** user's home directory.

To override these defaults, you must provide a master configuration file. The default master configuration file name is **/etc/LoadL.cfg**.

You can also override the location of the master configuration file by specifying the **LOADL_CONFIG** environment variable. For an example of when you might want to do this, see "Querying multiple LoadLeveler clusters" on page 74. This cannot be used to switch between two database-based LoadLeveler configurations, because **/etc/xcat/cfgloc** is also used to determine the dataspace name.

Setting the LoadLeveler user

You can override the default LoadLeveler user name and group name by specifying the following keywords in the master configuration file:

LoadLUserid
Specifies the LoadLeveler user ID.

LoadLGroupid
Specifies the LoadLeveler group ID.

Notes:

1. If you change the LoadLeveler user ID to something other than **loadl**, and you are using configuration files as the source for configuration data, you will have to make sure your configuration files are owned by this ID.

2. If Cluster Security (CtSec) services is enabled, make sure you update the **unix.map** file if the **LoadLUserid** is specified as something other than **loadl**. Refer to "Steps for enabling CtSec services" on page 62 for more details.

Setting the configuration source

LoadLeveler configuration data is obtained in one of three ways, depending on the content of the **/etc/LoadL.cfg** file:

1. By reading the LoadLeveler configuration database tables
2. By reading LoadLeveler configuration files
3. By fetching the configuration data from a LoadLeveler daemon on a different machine

One of the following keywords, **LoadLConfig**, **LoadLConfigHosts**, and **LoadLDB** can be specified in the master configuration file to designate which method will be used on a machine to obtain LoadLeveler configuration data:

LoadLConfig
Specifies the full path name of the global configuration file which identifies the set of configuration files. The set consists of the global configuration file, the administration file, and the local configuration file. The global configuration file must contain the location of the administration file, and may optionally contain the location of a local configuration file.

Syntax:

```
LoadLConfig = global configuration file path
```

LoadLConfigHosts
This keyword is used to designate one or more hosts that will serve LoadLeveler configuration data. LoadLeveler processes will attempt to retrieve configuration data from the hosts in the order they are listed. The hosts specified in this list must be configured to use a database for LoadLeveler configuration data.

Syntax:

```
LoadLConfigHosts = host1 host2 .. hostn
```

LoadLDB
This keyword designates an **odbc.ini** stanza name identifying the database, or data source name (DSN), to be used for LoadLeveler configuration data. The stanza used must be defined in the **/etc/odbc.ini** file. When setting up the MySQL xCAT database, set up a stanza in **/etc/odbc.ini** and specify the stanza name in the **LoadLDB** statement.

Syntax:

```
LoadLDB = xcatdb
```

If none of the keywords, **LoadLConfig**, **LoadLConfigHosts**, or **LoadLDB**, are specified, then by default, LoadLeveler will expect to find a global configuration file in the home directory of the LoadLeveler user ID, and will obtain configuration data from configuration files.

Overriding the shared memory key

LoadLeveler uses a shared memory segment to cache parsed configuration data read from the LoadLeveler configuration database tables or from LoadLeveler configuration and administration files. LoadLeveler uses a key to uniquely identify

its shared memory segment. Normally a default shared memory segment key generated by LoadLeveler is sufficient for this purpose. However, there is a small possibility that another application, running on the same nodes as LoadLeveler, will use the same shared memory segment key. In this case, LoadLeveler will generate a non-default key, and store it in the master configuration file, using the **LoadLConfigShmKey** keyword.

This keyword can also be used by the LoadLeveler administrator to override the default for the shared memory segment key.

LoadLConfigShmKey
Records the value used for the shared memory segment key when it is not possible to use the generated default key because of a conflict with another shared memory application.

Syntax:

```
LoadLConfigShmKey = number
```

All LoadLeveler processes that need to access configuration data will first read the master configuration file. If this keyword is specified, the process will use the specified key to access the LoadLeveler shared memory segment. If this keyword is not specified, a default key will be generated.

Note: The shared memory segment key does not have to be the same on all nodes of the LoadLeveler cluster.

Default value: No default value is set.

File-based configuration

LoadLeveler uses file-based configuration by default. For file-based configuration, the master configuration file is optional. Three types of files are used to define the configuration:

- *Global configuration file:* This file, by default, is called the **LoadL_config** file and it contains configuration information common to all nodes in the LoadLeveler cluster. Use the **LoadLConfig** keyword in the master configuration file to specify the location and name of the file if you do not want to use the default.
- *Local Configuration File:* This file is generally called **LoadL_config.local** (although it is possible for you to rename it). This file contains specific configuration information for an individual node. The **LoadL_config.local** file is in the same format as **LoadL_config** and the information in this file overrides any information specified in **LoadL_config**. It is an optional file that you use to modify information on a local machine. Its full path name is specified in the **LoadL_config** file by using the **LOCAL_CONFIG** keyword.
- *Administration file:* The administration file optionally lists and defines the machines in the LoadLeveler cluster and the characteristics of classes, users, and groups. Its full path name is specified in the **LoadL_config** file by using the **ADMIN_FILE** keyword.

To easily manage LoadLeveler, you should have one global configuration file and only one administration file, both centrally located on a machine in the LoadLeveler cluster. Every other machine in the cluster must be able to read the configuration and administration file that are located on the central machine.

You may have multiple local configuration files that specify information specific to individual machines.

LoadLeveler does not prevent you from having multiple copies of administration files, but you need to be sure to update all the copies whenever you make a change to one. Having only one administration file prevents any confusion.

Database configuration option

When LoadLeveler is part of the software stack in a cluster managed by xCAT, LoadLeveler's configuration can share xCAT's MySQL or DB2® database space. In this case, the master configuration file is required and the **LoadLDB** indicates the data source name.

For more information, see:

- *LoadLeveler for AIX: Installation Guide*
- *LoadLeveler for Linux: Installation Guide*

Or, go to *Setting Up MySQL as the xCAT DB* or *Setting Up DB2 as the xCAT DB* at XCAT Documentation (http://sourceforge.net/apps/mediawiki/xcat/index.php?title=XCAT_Documentation).

To initialize the configuration, run the **llconfig** or **llrconfig** command with the **-i** option. You must specify configuration files as input to this command. Administration files cannot be specified in the list. Administration files are specified by the **ADMIN_FILE** keyword in the configuration file. All of the data from the configuration files that are used for initialization are stored in the database for the default machine. To specify configuration settings for individual machines, you must update the configuration after initialization.

You can manage the database-based configuration by using the **llconfig** or **llrconfig** options or LoadLeveler's configuration editor to display and change the values of keywords and stanzas in the database.

Understanding remotely configured nodes

When LoadLeveler is configured to use a database to contain LoadLeveler configuration data, it is possible to configure "remotely configured nodes" that do not have access to the database. This kind of configuration permits an installation to define a LoadLeveler cluster without having to provide database clients on all nodes.

The remotely configured nodes are configured by specifying the **LoadLConfigHosts** keyword in the master configuration file. This keyword is used to specify the host, or hosts, that can serve configuration data to the remotely configured node. For more information about the **LoadLConfigHosts** keyword, see “Setting the configuration source” on page 41.

There are limitations to LoadLeveler nodes that are configured in this way because they do not have direct access to the configuration data in the database and are dependent on another node for configuration data.

Not all LoadLeveler daemons can run on remotely configured nodes. It is expected that remotely configured nodes would be configured to run the **startd** daemon (and starter process) and optionally the keyboard daemon. The central manager and resource manager depend on having database access and may not be configured on a remotely configured node. Although the regional manager and

schedd daemon may be configured on a remotely configured node, it is not recommended, because the use of a database for these daemons could change in the future.

There is a situation where LoadLeveler processes (daemons, commands or APIs) invoked on a remotely configured node may not always work if a configuration server is not available. A shared memory segment is used to cache configuration data. Normally, LoadLeveler processes will be able to read LoadLeveler configuration data from the shared memory segment. If the shared memory segment does not already exist, the configuration server specified by the **LoadLConfigHosts** statement must be contacted to obtain the configuration data to store in the shared memory segment. In this situation, if the configuration servers cannot be contacted, the LoadLeveler process will fail.

The shared memory segment is usually created when the first LoadLeveler process runs, and is subsequently available for use by other LoadLeveler processes. So normally you can expect to run LoadLeveler processes on remotely configured nodes. The shared memory segment can be removed by issuing the **llctl rmshm** command. The LoadLeveler shared memory segment is also removed when uninstalling LoadLeveler and when installing updates to the LoadLeveler software.

There are two command options that always rely on the availability of a configuration server. The **llctl start** and **reconfig** command options depend on the configuration server regardless of whether a shared memory segment exists or not. The **start** or **reconfig** option must access the most current database configuration data in order to refresh the configuration data in the shared memory segment. If a configuration server cannot be contacted in this situation, the command cannot perform its function. To ensure configuration server machines are started or reconfigured before the machines that depend on these servers, use **xdsh** to run the commands.

Using the configuration editor

To make it easier to view, update, and maintain the LoadLeveler configuration using the database option, a form-based configuration editor is provided for administrators. This tool allows the administrator to modify configuration attributes, make updates into the configuration database, and invoke reconfiguration for just the nodes that need to pick up the changed configuration.

In the database, the configuration is kept in several tables and the configuration editor groups input for the tables to make it easier for the administrator to set up and update the configuration. The editor consists of several tabbed panels containing forms that present one or more configuration database tables and a search panel to help the administrator find the panel in which a keyword can be updated.

For information about setting up the configuration editor, see the “Software requirements” topic in the *LoadLeveler for AIX: Installation Guide* or the *LoadLeveler for Linux: Installation Guide*.

To invoke the editor, point your browser to the following URL:

http://your_server_machine/ll/llconfig_editor.pl

where your_server_machine is the node where you are running the HTTP Server.

You will need to login to the configuration editor using a database ID and password with access to make changes in the database.

To use the editor, update the fields in the forms and click the **Add**, **Update**, or **Delete** buttons to update the configuration change in the database. After you have completed all the changes you will make for the session, make sure that you click the **Reconfigure** button to reconfigure LoadLeveler so the changes are effective.

If you cannot find a keyword you want to update, use the Search panel to find the panel containing a keyword. You may enter all or part of the keyword you are looking for.

Modifying configuration data

By taking a look at the configuration files that come with LoadLeveler, you will find that there are many parameters that you can set. In most cases, you will only have to modify a few of these parameters.

In some cases, though, depending upon the LoadLeveler nodes, network connection, and hardware availability, you may need to modify additional parameters.

All LoadLeveler commands, daemons, and processes read the configuration data at start up time. To ensure that the configuration for all LoadLeveler commands, daemons, and processes are in sync, run the reconfiguration command on all machines in the cluster each time the configuration changes.

Defining LoadLeveler administrators

Specify the **LOADL_ADMIN** keyword with a list of user names of those individuals who will have administrative authority.

These users are able to invoke the administrator-only commands such as **llctl**, **llfavorjob**, and **llfavoruser**.

LoadLeveler administrators on this list also receive mail describing problems that are encountered by the master daemon. When CtSec is enabled, the **LOADL_ADMIN** list is used only as a mailing list. For more information, see "Defining security mechanisms" on page 60.

For file-based configuration, an administrator on a machine is granted administrative privileges on that machine. An administrator is not granted administrative privileges on other machines. To be an administrator on all machines in the LoadLeveler cluster, either specify your user ID in the global configuration file with no entries in the local configuration file, or specify your user ID in every local configuration file that exists in the LoadLeveler cluster.

When using the database configuration option, the list of administrators applies to all machines in a cluster.

For information about configuration keyword syntax and other details, see Chapter 10, "Configuration keyword reference," on page 231.

Defining a LoadLeveler cluster

It will be necessary to define the characteristics of the LoadLeveler cluster.

Table 9 lists the topics that discuss how you can define the characteristics of the LoadLeveler cluster.

Table 9. Roadmap for defining LoadLeveler cluster characteristics

To learn about:	Read the following:
Defining characteristics of specific LoadLeveler daemons	• "Choosing a scheduler" • "Setting negotiator characteristics and policies" on page 47 • "Specifying alternate central managers" on page 48
Defining other cluster characteristics	• "Defining network characteristics" on page 49 • "Specifying file and directory locations" on page 49 • "Configuring recording activity and log files" on page 52 • "Setting up file system monitoring" on page 58
Correctly specifying configuration keywords	Chapter 10, "Configuration keyword reference," on page 231

For information on working on with daemons and machines in a LoadLeveler cluster, see the **llctl** and **llinit** commands in *LoadLeveler: Command and API Reference*.

Choosing a scheduler

This topic discusses the types of schedulers available, which you may specify using the configuration keyword **SCHEDULER_TYPE**.

For information about the configuration keyword syntax and other details, see Chapter 10, "Configuration keyword reference," on page 231.

LL_DEFAULT
This scheduler runs serial jobs. It efficiently uses CPU time by scheduling jobs on what otherwise would be idle nodes (and workstations). It does not require that users set a wall clock limit. Also, this scheduler starts, suspends, and resumes jobs based on workload.

BACKFILL
This scheduler runs both serial and parallel jobs. The objective of BACKFILL scheduling is to maximize the use of resources to achieve the highest system efficiency, while preventing potentially excessive delays in starting jobs with large resource requirements. These large jobs can run because the BACKFILL scheduler does not allow jobs with smaller resource requirements to continuously use up resource before the larger jobs can accumulate enough resource to run.

The BACKFILL scheduler supports:
- The scheduling of multiple tasks per node
- The scheduling of multiple user space tasks per adapter
- The preemption of jobs
- The use of reservations
- The scheduling of inbound and outbound data staging tasks

These functions are not supported by the default LoadLeveler scheduler.

For more information about the BACKFILL scheduler, see "Using the BACKFILL scheduler" on page 114.

API This keyword option allows you to enable an external scheduler. The API

option is intended for installations that want to create a scheduling algorithm for parallel jobs based on site-specific requirements.

For more information about external schedulers, see "Using an external scheduler" on page 119.

Setting negotiator characteristics and policies

You may set the following negotiator characteristics and policies.

For information about configuration keyword syntax and other details, see Chapter 10, "Configuration keyword reference," on page 231.

- Prioritize the queue maintained by the negotiator

  Each job step submitted to LoadLeveler is assigned a system priority number, based on the evaluation of the **SYSPRIO** keyword expression. For file-based configuration this is found in the configuration file of the central manager. For database-based configuration, it is set for a cluster. The LoadLeveler system priority number is assigned to a new job step by the central manager when the job step is submitted to the central manager. Once assigned, the system priority number for a job step is not changed, except under the following circumstances:
  - An administrator or user issues the **llprio** command to change the system priority of the job step.
  - The value set for the **NEGOTIATOR_RECALCULATE_SYSPRIO_INTERVAL** keyword is not zero.
  - An administrator uses the **llmodify** command with the **-s** option to alter the system priority of a job step.
  - A program with administrator credentials uses the **ll_modify** subroutine to alter the system priority of a job step.

  Job steps assigned higher **SYSPRIO** numbers are considered for dispatch before job steps with lower numbers.

  For related information, see the following topics:
  - "Controlling the central manager scheduling cycle" on page 76
  - "Setting and changing the priority of a job" on page 224
  - **llmodify** command in *LoadLeveler: Command and API Reference*
  - **ll_modify** subroutine in *LoadLeveler: Command and API Reference*
- Prioritize the order of executing machines maintained by the negotiator

  Each executing machine is assigned a machine priority number, based on the evaluation of the **MACHPRIO** keyword expression in the configuration file of the central manager for file-based configuration or the **MACHPRIO** keyword expression for the cluster for database-based configuration. The LoadLeveler machine priority number is updated every time the central manager updates its machine data. Machines assigned higher **MACHPRIO** numbers are considered to run jobs before machines with lower numbers. For example, a machine with a **MACHPRIO** of 10 is considered to run a job before a machine with a **MACHPRIO** of 5. Similarly, a machine with a **MACHPRIO** of -2 would be considered to run a job before a machine with a **MACHPRIO** of -3.

  For the file-based configuration, note that the **MACHPRIO** keyword is valid only on the machine where the central manager is running. Using this keyword in a local configuration file has no effect.

  When you use a **MACHPRIO** expression that is based on load average, the machine may be temporarily ordered later in the list immediately after a job is scheduled to that machine. This temporary drop in priority happens because the negotiator adds a compensating factor to the startd machine's load average every time the negotiator assigns a job. For more information, see the **NEGOTIATOR_LOADAVG_INCREMENT** keyword in "Configuration keyword descriptions" on page 233.

- Specify additional negotiator policies

  This topic lists keywords that were not mentioned in the previous configuration steps. Unless your installation has special requirements for any of these keywords, you can use them with their default settings.
  - **NEGOTIATOR_INTERVAL**
  - **NEGOTIATOR_CYCLE_DELAY**
  - **NEGOTIATOR_CYCLE_TIME_LIMIT**
  - **NEGOTIATOR_LOADAVG_INCREMENT**
  - **NEGOTIATOR_PARALLEL_DEFER**
  - **NEGOTIATOR_PARALLEL_HOLD**
  - **NEGOTIATOR_RECALCULATE_SYSPRIO_INTERVAL**
  - **NEGOTIATOR_REJECT_DEFER**
  - **NEGOTIATOR_REMOVE_COMPLETED**
  - **NEGOTIATOR_RESCAN_QUEUE**

Specifying alternate central managers

In a keyword statement in the configuration, you specified which machine would serve as the central manager. You can also assign one or more alternate central managers in case network communication, software, or hardware failures make the primary central manager unusable; to do so, specify one or more alternate central managers by adding the following keyword statement to the configuration file or update the keyword in the configuration database:

```
CENTRAL_MANAGER_LIST = primary_central_manager_machine \
                       [alternate_central_manager_machine_list]
```

If the primary central manager fails, the alternate central manager then becomes the central manager. The alternate central manager is chosen based upon the order in which its name appears, either in the machine stanza in the administration file or in the alternate central manager machine list specified in the configuration file or database.

Note: For file-based configuration, the primary and alternate central manager machines specified in the configuration file override those specified in the administration file.

When an alternate becomes the central manager, jobs will not be lost, but it may take a few minutes for all of the machines in the cluster to check in with the new central manager. As a result, job status queries may be incorrect for a short time.

An alternate central manager will not receive information on jobs that were running on a machine or are managed by a job manager on a machine that is either down or cannot communicate with the alternate central manager. For example, if a job manager is running on the same machine as the primary central manager and the machine goes down, when the alternate central manager takes over, it will have no information about the jobs managed by the job manager on the primary central manager machine. In this case, job status queries will not include information on those jobs.

When you define alternate managers, you should set the following keywords in the configuration:
- **FAILOVER_HEARTBEAT_INTERVAL**
- **FAILOVER_HEARTBEAT_RETRIES**

In the following example, the alternate central manager will wait for 30 intervals, where each interval is 45 seconds:

```
# Set a 45 second interval
FAILOVER_HEARTBEAT_INTERVAL = 45
# Set the number of intervals to wait
FAILOVER_HEARTBEAT_RETRIES = 30
```

For more information on central manager backup, refer to "What happens if the central manager isn't operating?" on page 399. For information about configuration keyword syntax and other details, see Chapter 10, "Configuration keyword reference," on page 231.

Specifying alternate resource managers

In a keyword statement in the configuration file or database, you specified which machine would serve as the resource manager. You can also assign one or more alternate resource managers in case network communication, software, or hardware failures make the primary resource manager unusable; to do so, ensure that your keyword statement in the configuration file or configuration database includes a list of alternate resource managers:

```
RESOURCE_MGR_LIST = primary_resource_manager_machine \
                    [alternate_resource_manager_machine_list]
```

If the primary resource manager fails, the alternate resource manager then becomes the resource manager. The alternate resource manager is chosen based upon the order in which its name appears in the alternate resource manager machine list specified in the configuration file or database.

For information about configuration keyword syntax and other details, see Chapter 10, "Configuration keyword reference," on page 231.

Specifying alternate region managers

In a keyword statement in the region stanza, you specified which machine would serve as the region manager. You can also assign one or more alternate region managers in case network communication, software, or hardware failures make the primary region manager unusable; to do so, ensure that your keyword statement in the region stanza includes a list of alternate region managers:

```
region_mgr_list = primary_resource_manager_machine \
                  [alternate_resource_manager_machine_list]
```

If the primary region manager fails, the alternate region manager then becomes the region manager. The alternate region manager is chosen based upon the order in which its name appears in the alternate region manager machine list specified in the region stanza.

For information, see "Defining regions" on page 106.

Defining network characteristics

A **port number** is an integer that specifies the port to use to connect to the specified daemon. You can define these port numbers in the configuration keyword or the **/etc/services** file or you can accept the defaults. LoadLeveler first looks in the configuration for these port numbers. If LoadLeveler does not find the value in the configuration, it looks in the **/etc/services** file. If the value is not found in this file, the default is used.

See Appendix B, "LoadLeveler port usage," on page 417 for more information.

Specifying file and directory locations

The sample configuration file provided with LoadLeveler specifies default locations for all of the files and directories. You can modify their locations using the

keywords shown in Table 10. Keep in mind that the LoadLeveler installation process installs files in these directories and these files may be periodically cleaned up. Therefore, you should not keep any files that do not belong to LoadLeveler in these directories.

Managing distributed software systems is a primary concern for all system administrators. Allowing users to share file systems to obtain a single, network-wide image, is one way to make managing LoadLeveler easier.

Table 10. Default locations for all of the files and directories

To specify the location of the:	**Specify this keyword:**
Administration file	**ADMIN_FILE**
Local configuration file	**LOCAL_CONFIG**
Local directory	The following subdirectories reside in the local directory. It is possible that the local directory and LoadLeveler's home directory are the same. • **COMM** • **EXECUTE** • **LOG** • **SPOOL** and **HISTORY** **Tip:** To maximize performance, you should keep the log, spool, and execute directories in a local file system. Also, to measure the performance of your network, consider using one of the available products, such as Toolbox/6000. On clusters with diskless nodes, the execute directory should be located in RAM disk. When using a local file system for the spool and history directories, commands such as **llmovespool** and **llacctmrg** will not have access to the data in those directories if a machine is down or not accessible on the network.
Release directory	When a LoadLeveler daemon or process needs to start another command or process, it will use the following keywords to locate the appropriate binary. If you change the location of the LoadLeveler release directory you must change these keywords to point to the new location. **RELEASEDIR** Is the directory created during installation to contains the **bin**, **lib**, and **include** directories for the LoadLeveler code. **BIN** Is a subdirectory in the release directory that is created during installation for the binaries for the LoadLeveler daemons and commands.

Table 10. Default locations for all of the files and directories (continued)

To specify the location of the:	Specify this keyword:
Core dump directory	You may specify alternate directories to hold core dumps for the daemons and starter process: • **MASTER_COREDUMP_DIR** • **NEGOTIATOR_COREDUMP_DIR** • **RESOURCE_MGR_COREDUMP_DIR** • **REGION_MGR_COREDUMP_DIR** • **SCHEDD_COREDUMP_DIR** • **STARTD_COREDUMP_DIR** • **STARTER_COREDUMP_DIR** • **KBDD_COREDUMP_DIR** When specifying core dump directories, be sure that the access permissions are set so the LoadLeveler daemon or process can write to the core dump directory. The permissions set for path names specified in the keywords just mentioned must allow writing by both root and the LoadLeveler ID. The permissions set for the path name specified for the STARTER_COREDUMP_DIR keyword must allow writing by root, the LoadLeveler ID, and any user who can submit LoadLeveler jobs. The simplest way to be sure the access permissions are set correctly is to set them the same as are set for the **/tmp** directory. If a problem with access permissions prevents a LoadLeveler daemon or process from writing to a core dump directory, then a message will be written to the log, and the daemon or process will continue using the default **/tmp** directory for core files.

For information about configuration keyword syntax and other details, see Chapter 10, "Configuration keyword reference," on page 231.

Locating LoadLeveler binaries when the scheduler component is not installed

If your cluster will only run the resource manager or it will include nodes where only the resource manager component is installed, you will need to make changes to your configuration for locating the LoadLeveler binaries. The **RELEASEDIR** and **BIN** keywords are used for defining locations.

If the cluster will only run the LoadLeveler resource manager and if you use the sample file **LoadL_config**, then do *one* of the following:

Change:

```
RELEASEDIR  = /usr/lpp/LoadL/full
```

to

```
RELEASEDIR  = /usr/lpp/LoadL/resmgr/full
```

or:

Create the following link on each node:

```
ln -s /usr/lpp/LoadL/resmgr/full /usr/lpp/LoadL/full
```

or:

Run the **llrinit** command on each of the nodes to create the link.

If the cluster consists of both resource manager and scheduler nodes, but some nodes have just the resource manager component installed, use the **LoadL_config.l** sample file to create your configuration. This sample uses **RELEASEDIR= /usr/lpp/LoadL/resmgr/full** and specifies the scheduler full path for those binaries that need it.

Configuring recording activity and log files

The LoadLeveler daemons and processes keep log files according to the specifications in the configuration file or database. Administrators can also configure the LoadLeveler daemons to store additional debugging messages in a circular buffer in memory. A number of keywords are used to describe where LoadLeveler maintains the logs and how much information is recorded in each log and buffer. These keywords, shown in Table 11, are repeated in similar form to specify the path name of the log file, its maximum length, the size of the circular buffer, and the debug flags to be used for the log and the buffer.

"Controlling the logging buffer" on page 53 describes how administrators can configure LoadLeveler to buffer debugging messages.

"Controlling debugging output" on page 55 describes the events that can be reported through logging controls.

"Saving log files" on page 58 describes the configuration keyword to use to save logs for problem diagnosis.

For information about configuration keyword syntax and other details, see Chapter 10, "Configuration keyword reference," on page 231.

Table 11. Log control statements

Daemon/ Process	**Log File** *(required)* **(See note 1 on page 53)**	**Max Length** *(required)* **(See note 3 on page 53)**	**Debug Control** *(required)* **(See note 5 on page 53)**
Master	**MASTER_LOG =** *path*	**MAX_MASTER_LOG =** *bytes* [*buffer bytes*]	**MASTER_DEBUG =** *flags* [*buffer flags*]
Schedd	**SCHEDD_LOG =** *path*	**MAX_SCHEDD_LOG =** *bytes* [*buffer bytes*]	**SCHEDD_DEBUG =** *flags* [*buffer flags*]
Startd	**STARTD_LOG =** *path*	**MAX_STARTD_LOG =** *bytes* [*buffer bytes*]	**STARTD_DEBUG =** *flags* [*buffer flags*]
Starter	**STARTER_LOG** = *path*	**MAX_STARTER_LOG =** *bytes* [*buffer bytes*]	**STARTER_DEBUG =** *flags* [*buffer flags*]
Negotiator	**NEGOTIATOR_LOG =** *path*	**MAX_NEGOTIATOR_LOG =** *bytes* [*buffer bytes*]	**NEGOTIATOR_DEBUG =** *flags* [*buffer flags*]
Kbdd	**KBDD_LOG =** *path*	**MAX_KBDD_LOG =** *bytes* [*buffer bytes*]	**KBDD_DEBUG =** *flags* [*buffer flags*]
Region_mgr	**REGION_MGR_LOG =** *path*	**MAX_REGION_MGR_LOG =** *bytes* [*buffer bytes*]	**REGION_MGR_DEBUG =** *flags* [*buffer flags*]
Resource_mgr	**RESOURCE_MGR_LOG =** *path*	**MAX_RESOURCE_MGR_LOG =** *bytes* [*buffer bytes*]	**RESOURCE_MGR_DEBUG =** *flags* [*buffer flags*]

where:

buffer bytes
Is the size of the circular buffer. The default value is 0, which indicates that the buffer is disabled. To prevent the daemon from running out of memory, this value should not be too large. Brackets must be used to specify buffer bytes.

buffer flags
Indicates that messages with *buffer flags* in addition to messages with *flags* will be stored in the circular buffer in memory. The default value is blank, which indicates that the logging buffer is disabled because no additional debug flags were specified for buffering. Brackets must be used to specify buffer flags.

Notes:

1. When coding the *path* for the log files, it is not necessary that all LoadLeveler daemons keep their log files in the same directory, however, you will probably find it a convenient arrangement.
2. When the database configuration option is used, the flags and buffer flags strings are limited to 255 characters. Flags or buffer flags strings longer than 255 characters will be truncated.
3. There is a maximum length, in bytes, beyond which the various log files cannot grow. Each file is allowed to grow to the specified length and is then saved to an **.old** file. The **.old** files are overwritten each time the log is saved, thus the maximum space devoted to logging for any one program will be twice the maximum length of its log file. The default length is 64 KB. To obtain records over a longer period of time, that do not get overwritten, you can use the **SAVELOGS** keyword. See "Saving log files" on page 58 for more information on extended capturing of LoadLeveler logs.

   You can also specify that the log file be started anew with every invocation of the daemon by setting the **TRUNC** statement to **true** as follows:
   - **TRUNC_MASTER_LOG_ON_OPEN =** true | **false**
   - **TRUNC_STARTD_LOG_ON_OPEN =** true | **false**
   - **TRUNC_SCHEDD_LOG_ON_OPEN =** true | **false**
   - **TRUNC_KBDD_LOG_ON_OPEN =** true | **false**
   - **TRUNC_STARTER_LOG_ON_OPEN =** true | **false**
   - **TRUNC_NEGOTIATOR_LOG_ON_OPEN =** true | **false**
   - **TRUNC_REGION_MGR_LOG_ON_OPEN =** true | **false**
   - **TRUNC_RESOURCE_MGR_LOG_ON_OPEN =** true | **false**
4. LoadLeveler creates temporary log files used by the **starter** daemon. These files are used for synchronization purposes. When a job starts, a **StarterLog.***pid* file is created. When the job ends, this file is appended to the **StarterLog** file.
5. Normally, only those who are installing or debugging LoadLeveler will need to use the debug flags, described in "Controlling debugging output" on page 55 The default error logging, obtained by leaving the right side of the debug control statement null, will be sufficient for most installations.

Controlling the logging buffer:
LoadLeveler allows a LoadLeveler daemon to store log messages in a buffer in memory instead of writing the messages to a log file. The administrator can force the messages in this buffer to be written to the log file, when necessary, to diagnose a problem. The buffer is circular and once it is full, older messages are discarded as new messages are logged. The **llctl dumplogs** command is used to write the contents of the logging buffer or locking records to the appropriate log file for the **Master**, **Negotiator**, **Schedd**, **Startd**, **Resource Manager**, and **Region Manager** daemons.

Buffering will be disabled if either the buffer length is 0 or no additional debug flags are specified for buffering.

See “Configuring recording activity and log files” on page 52 for log control statement specifications. See *LoadLeveler: Diagnosis and Messages Guide* for additional information on LoadLeveler log files.

Logging buffer example:

With the following configuration, the **Schedd** daemon will write only **D_ALWAYS** and **D_SCHEDD** messages to the ${LOG}/SchedLog log file. The following messages will be stored in the buffer:

- **D_ALWAYS**
- **D_SCHEDD**
- **D_FULLDEBUG**

The maximum size of the Schedd log is 64 MB and the size of the logging buffer is 32 MB.

```
SCHEDD_LOG = ${LOG}/SchedLog
MAX_SCHEDD_LOG = 64000000 [32000000]
SCHEDD_DEBUG = D_SCHEDD [D_FULLDEBUG]
```

To write the contents of the logging buffer to the SchedLog file on the machine, issue:

```
llctl dumplogs buffer
```

To write the contents of the logging buffer to the SchedLog file on **node1** in the LoadLeveler cluster, issue:

```
llctl -h node1 dumplogs buffer
```

To write the contents of the logging buffers to the SchedLog files on all machines, issue:

```
llctl -g dumplogs buffer
```

Note that the messages written from the logging buffer include a bracket message and a prefix to identify them easily.

```
======================BUFFER BEGIN=======================

BUFFER: message .....
BUFFER: message .....

======================BUFFER END=========================
```

Controlling locking records:

To turn on the locking function of a daemon, specify the **D_LOCK_TRACE** debugging flag.

Note: D_LOCK_TRACE messages are not handled in the same way as other debug flags are handled. Rather than being printed to the logs as they occur, these messages are kept in memory and updated when there are locking events like getting a new lock request or releasing a lock. Use **llctl dumplogs locks** to print messages to the logs about locks currently pending or held.

Locking records examples:

To save the locking records in the LoadLeveler daemons to the corresponding existing log files in the LOG directory of node c197blade7b02, issue:

```
llctl -h c197blade7b02 dumplogs locks
```

With the following configuration, the locking records of the Schedd daemon will be stored.

```
SCHEDD_DEBUG = D_LOCK_TRACE
```

To write the locking records to the SchedLog file on the machine, issue:

```
llrctl dumplogs locks
```

To write the locking records to the SchedLog file on node1 in the LoadLeveler cluster, issue:

```
llctl –h node1 dumplogs lock
```

Note that the locking records are grouped within BEGIN and END lines to make them easy to identify in the log file.

```
============ LOCKING RECORDS BEGIN ============

LOCKID   TID  STAT  TYPE   SC   TIMESTAMP  LINE FUNCTION | DESCRIPTION
                                                         ...
                                                         ...
============ LOCKING RECORDS END ============
```

Controlling debugging output:
You can control the level of debugging output logged by LoadLeveler programs.

The following flags are presented here for your information, though they are used primarily by IBM personnel for debugging purposes:

D_ACCOUNT
: Logs accounting information about processes. If used, it may slow down the network.

D_ACCOUNT_DETAIL
: Logs detailed accounting information about processes. If used, it may slow down the network and increase the size of log files.

D_ADAPTER
: Logs messages related to adapters.

D_AFS
: Logs information related to AFS credentials.

D_CKPT
: Logs information related to checkpoint and restart.

D_CONFIG
: Displays messages detailing configuration processing during the start up or reconfiguration of a LoadLeveler process.

D_DAEMON
: Logs information regarding basic daemon set up and operation, including information on the communication between daemons.

D_DATABASE
: Logs information pertaining to database interactions.

D_DISPATCHING_CYCLE
: Traces the dispatching cycle from the cluster's startup. This flag is set using the **TRACE** configuration keyword and cannot be set as a ***_DEBUG** flag. See the **TRACE** keyword on page 280 for more information.

D_EXPR
: Logs steps in parsing and evaluating control expressions.

D_FAIRSHARE
: Displays messages related to fair share scheduling in the daemon logs. In

the global configuration file, **D_FAIRSHARE** can be added to **SCHEDD_DEBUG** and **NEGOTIATOR_DEBUG**.

D_FULLDEBUG
Logs details about most actions performed by each daemon but doesn't log as much activity as setting all the flags.

D_HIERARCHICAL
Used to enable messages relating to problems related to the transmission of hierarchical messages. A hierarchical message is sent from an originating node to lower ranked receiving nodes.

D_JOB
Logs job requirements and preferences when making decisions regarding whether a particular job should run on a particular machine.

D_JOB_LIFECYCLE
Enables the administrator to control whether requests for job lifecycle traces will be granted. This flag is set using the **TRACE** configuration keyword and cannot be set as a ***_DEBUG** flag. See the **TRACE** keyword on page 280 for more information.

D_KERNEL
Activates diagnostics for errors involving the process tracking kernel extension.

D_LOAD
Displays the load average on the startd machine.

D_LOCKING
Logs requests to acquire and release locks. This keyword is deprecated in favor of the **D_LOCK_TRACE** keyword.

D_LOCK_TRACE
Activates the lock tracing function. This flag tracks locking events internally in memory and keeps only events for locks not yet released.

The **llctl dumplogs locks** command is used to move the locking records from memory into the logs (see *LoadLeveler: Command and API Reference* for more information about the **llctl** command).

D_LXCPUAFNT
Logs messages related to Linux CPU affinity. This flag is only valid for the **startd** daemon.

D_MACHINE
Logs machine control functions and variables when making decisions regarding starting, suspending, resuming, and aborting remote jobs.

D_MUSTER
Logs information related to multicluster processing.

D_NEGOTIATE
Displays the process of looking for a job to run in the negotiator. It only pertains to this daemon.

D_ODBC_DETAIL
Logs information pertaining to the ODBC interface.

D_PCRED
Directs that extra debug should be written to a file if the `setpcred()` function call fails.

D_PERF
Logs performance-related information in LoadLeveler daemon logs, especially the negotiator and Schedd daemon logs. Turning on **D_PERF** will produce microsecond time stamps for each entry in the daemon log.

D_PERF_DETAIL
Logs detailed performance-related information in LoadLeveler daemon

logs, especially the negotiator and Schedd daemon logs. Turning on **D_PERF_DETAIL** will produce microsecond time stamps for each entry in the daemon log.

D_PROC
Logs information about jobs being started remotely such as the number of bytes fetched and stored for each job.

D_QUEUE
Logs changes to the job queue.

D_REFCOUNT
Logs activity associated with reference counting of internal LoadLeveler objects.

D_REGIONMGR
Displays how the region manager works internally.

D_RESERVATION
Logs reservation information in the negotiator and Schedd daemon logs. **D_RESERVATION** can be added to **SCHEDD_DEBUG** and **NEGOTIATOR_DEBUG**.

D_RESOURCE
Logs messages about the management and consumption of resources. These messages are recorded in the negotiator log.

D_SCHEDD
Displays how the Schedd works internally.

D_SDO
Displays messages detailing LoadLeveler objects being transmitted between daemons and commands.

D_SECURITY
Logs information related to Cluster Security (CtSec) services identities.

D_SPOOL
Logs information related to the usage of databases in the LoadLeveler **spool** directory.

D_STANZAS
Displays internal information about the parsing of the administration file.

D_STARTD
Displays how the startd works internally.

D_STARTER
Displays how the starter works internally.

D_STREAM
Displays messages detailing socket I/O.

D_SWITCH
Logs entries related to switch activity and LoadLeveler Switch Table Object data.

D_THREAD
Displays the ID of the thread producing the log message. The thread ID is displayed immediately following the date and time. This flag is useful for debugging threaded daemons.

D_XDR
Logs information regarding External Data Representation (XDR) communication protocols.

For example:

```
SCHEDD_DEBUG = D_SCHEDD D_XDR
```

Causes the job manager to log information about its internal workings and exchange xdr messages with other LoadLeveler daemons. These flags will primarily be of interest to LoadLeveler implementers and debuggers.

The **LL_COMMAND_DEBUG** environment variable can be set to a string of debug flags the same way as the ***_DEBUG** configuration keywords are set. Normally, LoadLeveler commands and APIs do not print debug messages, but with this environment variable set, the requested classes of debugging messages will be logged to stderr. For example:

```
LL_COMMAND_DEBUG="D_ALWAYS D_STREAM" llstatus
```

will cause the **llstatus** command to print out debug messages related to I/O to stderr.

Saving log files: By default, LoadLeveler stores only the two most recent iterations of a daemon's log file (<*daemon name*>Log, and <*daemon name*>Log.old).

Occasionally, for problem diagnosing, users will need to capture LoadLeveler logs over an extended period. Users can specify that all log files be saved to a particular directory by using the **SAVELOGS** keyword. Be aware that LoadLeveler does not provide any way to manage and clean out all of those log files, so users must be sure to specify a directory in a file system with enough space to accommodate them. This file system should be separate from the one used for the LoadLeveler log, spool, and execute directories.

Each log file is represented by the name of the daemon that generated it, the exact time the file was generated, and the name of the machine on which the daemon is running. When you list the contents of the **SAVELOGS** directory, the list of log file names looks like this:

```
NegotiatorLogNov02.16:10:39.123456.c163n10.ppd.pok.ibm.com
NegotiatorLogNov02.16:10:42.987654.c163n10.ppd.pok.ibm.com
NegotiatorLogNov02.16:10:46.564123.c163n10.ppd.pok.ibm.com
NegotiatorLogNov02.16:10:48.234345.c163n10.ppd.pok.ibm.com
NegotiatorLogNov02.16:10:51.123456.c163n10.ppd.pok.ibm.com
NegotiatorLogNov02.16:10:53.566987.c163n10.ppd.pok.ibm.com
StarterLogNov02.16:09:19.622387.c163n10.ppd.pok.ibm.com
StarterLogNov02.16:09:51.499823.c163n10.ppd.pok.ibm.com
StarterLogNov02.16:10:30.876546.c163n10.ppd.pok.ibm.com
SchedLogNov02.16:09:05.543677.c163n10.ppd.pok.ibm.com
SchedLogNov02.16:09:26.688901.c163n10.ppd.pok.ibm.com
SchedLogNov02.16:09:47.443556.c163n10.ppd.pok.ibm.com
SchedLogNov02.16:10:12.712680.c163n10.ppd.pok.ibm.com
SchedLogNov02.16:10:37.342156.c163n10.ppd.pok.ibm.com
StartLogNov02.16:09:05.697753.c163n10.ppd.pok.ibm.com
StartLogNov02.16:09:26.881234.c163n10.ppd.pok.ibm.com
StartLogNov02.16:09:47.231234.c163n10.ppd.pok.ibm.com
StartLogNov02.16:10:12.125556.c163n10.ppd.pok.ibm.com
StartLogNov02.16:10:37.961486.c163n10.ppd.pok.ibm.com
```

For information about configuration keyword syntax and other details, see Chapter 10, “Configuration keyword reference,” on page 231.

Setting up file system monitoring

You can use the file system keywords to monitor the file system space or inodes used by LoadLeveler for:

- Logs
- Saving executables
- Spool information
- History files

You can also use the file system keywords to take preventive action and avoid problems caused by running out of file system space or inodes. This is done by setting the frequency that LoadLeveler checks the file system free space or inodes

and by setting the upper and lower thresholds that initialize system responses to the free space available. By setting a realistic span between the lower and upper thresholds, you will avoid excessive system actions.

The file system monitoring keywords are:
- **FS_INTERVAL**
- **FS_NOTIFY**
- **FS_SUSPEND**
- **FS_TERMINATE**
- **INODE_NOTIFY**
- **INODE_SUSPEND**
- **INODE_TERMINATE**

For information about configuration keyword syntax and other details, see Chapter 10, "Configuration keyword reference," on page 231.

Defining LoadLeveler machine characteristics

You can use these keywords to define the characteristics of machines in the LoadLeveler cluster:
- **ARCH**
- **CLASS**
- **CUSTOM_METRIC**
- **CUSTOM_METRIC_COMMAND**
- **FEATURE**
- **MAX_STARTERS**
- **SCHEDD_RUNS_HERE**
- **SCHEDD_SUBMIT_AFFINITY**
- **STARTD_RUNS_HERE**
- **START_DAEMONS**
- **VM_IMAGE_ALGORITHM**
- **X_RUNS_HERE**

For information about configuration keyword syntax and other details, see Chapter 10, "Configuration keyword reference," on page 231.

Defining job classes that a LoadLeveler machine will accept

The following examples illustrate possible ways of defining job classes.

- **Example 1**

  This example specifies multiple classes:

  ```
  Class = No_Class(2)
  ```

  The machine will only run jobs that have either defaulted to or explicitly requested class **No_Class**. A maximum of two LoadLeveler jobs are permitted to run simultaneously on the machine if the **MAX_STARTERS** keyword is not specified. See "Specifying how many jobs a machine can run" on page 60 for more information on **MAX_STARTERS**.

- **Example 2**

  This example specifies multiple classes:

  ```
  Class = No_Class(1) Small(1) Medium(1) Large(1)
  ```

The machine will only run a maximum of four LoadLeveler jobs that have either defaulted to, or explicitly requested **No_Class**, **Small**, **Medium**, or **Large** class. A LoadLeveler job with class **IO_bound**, for example, would not be eligible to run here.

- **Example 3**

  This example specifies multiple classes:

  ```
  Class = B(2) D(1)
  ```

  The machine will run only LoadLeveler jobs that have explicitly requested class **B** or **D**. Up to three LoadLeveler jobs may run simultaneously: two of class **B** and one of class **D**. A LoadLeveler job with class **No_Class**, for example, would not be eligible to run here.

Specifying how many jobs a machine can run

To specify how many jobs a machine can run, you need to take into consideration both the **MAX_STARTERS** keyword and the **Class** statement. This is described in more detail in "Defining LoadLeveler machine characteristics" on page 59.

For example, if the configuration contains these statements:

```
Class = A(1) B(2) C(1)
MAX_STARTERS = 2
```

then the machine can run a maximum of two LoadLeveler jobs simultaneously. The possible combinations of LoadLeveler jobs are:

- A and B
- A and C
- B and B
- B and C
- Only A, or only B, or only C

If this keyword is specified together with a **Class** statement, the maximum number of jobs that can be run is equal to the lower of the two numbers. For example, if:

```
MAX_STARTERS = 2
Class = class_a(1)
```

then the maximum number of job steps that can be run is one (the **Class** statement defines one class).

If you specify **MAX_STARTERS** keyword without specifying a **Class** statement, by default one class still exists (called **No_Class**). Therefore, the maximum number of jobs that can be run when you do not specify a **Class** statement is one.

Note: If the **MAX_STARTERS** keyword is not defined in the configuration, the maximum number of jobs that the machine can run is equal to the number of classes in the **Class** statement.

Defining security mechanisms

LoadLeveler can be configured to control authentication and authorization of LoadLeveler functions by using Cluster Security (CtSec) services, a subcomponent of Reliable Scalable Cluster Technology (RSCT), which uses the host-based authentication (HBA) as an underlying security mechanism.

LoadLeveler permits only one security service to be configured at a time. You can skip this topic if you do not plan to use this security feature or if you plan to use

the credential forwarding provided by the **llgetdce** and **llsetdce** program pair. Refer to "Using the alternative program pair: llgetdce and llsetdce" on page 77 for more information.

LoadLeveler for Linux does not support CtSec security.

LoadLeveler can be enabled to interact with OpenSSL for secure multicluster communications

Table 12 lists the topics that explain LoadLeveler daemons and how you may define their characteristics and modify their behavior.

Table 12. Roadmap of configuration tasks for securing LoadLeveler operations

To learn about:	Read the following:
Securing LoadLeveler operations using cluster security services	• "Configuring LoadLeveler to use cluster security services" • "Steps for enabling CtSec services" on page 62 • "Limiting which security mechanisms LoadLeveler can use" on page 64
Enabling LoadLeveler to secure multicluster communication with OpenSSL	"Steps for securing communications within a LoadLeveler multicluster" on page 153
Correctly specifying configuration keywords	Chapter 10, "Configuration keyword reference," on page 231

Configuring LoadLeveler to use cluster security services

Cluster security (CtSec) services allows a software component to authenticate and authorize the identity of one of its peers or clients. When configured to use CtSec services, LoadLeveler will:

- Authenticate the identity of users and programs interacting with LoadLeveler.
- Authorize users and programs to use LoadLeveler services. It prevents unauthorized users and programs from misusing resources or disrupting services.

To use CtSec services, all nodes running LoadLeveler must first be configured as part of a cluster running Reliable Scalable Cluster Technology (RSCT). For details on CtSec services administration, see *IBM Reliable Scalable Cluster Technology: Administration Guide*, SA22-7889.

CtSec services are designed to use multiple security mechanisms and each security mechanism must be configured for LoadLeveler. At the present time, directions are provided only for configuring the host-based authentication (HBA) security mechanism for LoadLeveler's use. If CtSec is configured to use additional security mechanisms that are not configured for LoadLeveler's use, then the LoadLeveler configuration keyword **SEC_IMPOSED_MECHS** must be specified. This keyword is used to limit the security mechanisms that will be used by CtSec services to only those that are configured for use by LoadLeveler.

Authorization is based on user identity. When CtSec services are enabled for LoadLeveler, user identity will differ depending on the authentication mechanism in use. A user's identity in UNIX host-based authentication is the user's network identity which is comprised of the user name and host name, such as user_name@host.

LoadLeveler uses CtSec services to authorize owners of jobs, administrators and LoadLeveler daemons to perform certain actions. CtSec services uses its own identity mapping file to map the clients' network identity to a local identity when performing authorizations. A typical local identity is the user name without a hostname. The local identities of the LoadLeveler administrators must be added as members of the group specified by the keyword **SEC_ADMIN_GROUP**. The local identity of the LoadLeveler user name must be added as the sole member of the group specified by the keyword **SEC_SERVICES_GROUP**. The LoadLeveler Services and Administrative groups, those identified by the keywords **SEC_SERVICES_GROUP** and **SEC_ADMIN_GROUP**, must be the same across all nodes in the LoadLeveler cluster. To ensure consistency in performing tasks which require owner, administrative or daemon privileges across all nodes in the LoadLeveler cluster, user network identities must be mapped identically across all nodes in the LoadLeveler cluster. If this is not the case, LoadLeveler authorizations may fail.

Steps for enabling CtSec services:
It is necessary to enable LoadLeveler to use CtSec services. To enable LoadLeveler to use CtSec services, perform the following steps:

1. Include, in the Trusted Host List, the host names of all hosts with which communications may take place. If LoadLeveler tries to communicate with a host not on the Trusted Host List the message: The host identified in the credentials is not a trusted host on this system will occur. Additionally, the system administrator should ensure that public keys are manually exchanged between all hosts in the LoadLeveler cluster. Refer to *IBM Reliable Scalable Cluster Technology: Administration Guide*, SA22-7889 for information on setting up Trusted Host Lists and manually transferring public keys.
2. Create user IDs. Each LoadLeveler administrator and the LoadLeveler user ID need to be created, if they don't already exist. You can do this through SMIT or the **mkuser** command.
3. Ensure that the **unix.map** file contains the correct value for the service name **ctloadl** which specifies the LoadLeveler user name. If you have configured LoadLeveler to use **loadl** as the LoadLeveler user name, either by default or by specifying **loadl** in the **LoadLUserid** keyword of the **/etc/LoadL.cfg** file, nothing needs to be done. The default map file will contain the **ctloadl** service name already assigned to **loadl**. If you have configured a different user name in the **LoadLUserid** keyword of the **/etc/LoadL.cfg** file, you will need to make sure that the **/var/ct/cfg/unix.map** file exists and that it assigns the same user name to the **ctloadl** service name. If the **/var/ct/cfg/unix.map** file does not exist, create one by copying the default map file **/usr/sbin/rsct/cfg/unix.map**. Do not modify the default map file.

   If the value of the **LoadLUserid** and the value associated with **ctloadl** are not the same a security services error which indicates a UNIX identity mismatch will occur.
4. Add entries to the global mapping file of each machine in the LoadLeveler cluster to map network identities to local identities. This file is located at: **/var/ct/cfg/ctsec_map.global**. If this file doesn't yet exist, you should copy the default global mapping file to this location—don't modify the default mapping file. The default global mapping file, which is shared among all CtSec services exploiters, is located at **/usr/sbin/rsct/cfg/ctsec_map.global**. See *IBM Reliable Scalable Cluster Technology for AIX: Technical Reference*, SA22-78900 for more information on the mapping file.

   When adding names to the global mapping file, enter more specific entries ahead of the other, less specific entries. Remember that you must update the global mapping file on each machine in the LoadLeveler cluster, and each

mapping file has to be updated with the security services identity of each member of the **administrator** group, the **services** group, and the users. Therefore, you would have entries like this:

```
unix:brad@mach1.pok.ibm.com=bradleyf
unix:brad@mach2.pok.ibm.com=bradleyf
unix:brad@mach3.pok.ibm.com=bradleyf
unix:marsha@mach2.pok.ibm.com=marshab
unix:marsha@mach3.pok.ibm.com=marshab
unix:loadl@mach1.pok.ibm.com=loadl
unix:loadl@mach2.pok.ibm.com=loadl
unix:loadl@mach3.pok.ibm.com=loadl
```

However, if you're sure your LoadLeveler cluster is secure, you could specify mapping for all machines this way:

```
unix:brad@*=bradleyf
unix:marsha@*=marshab
unix:loadl@*=loadl
```

This indicates that the UNIX network identity of the users **brad**, **marsha** and **loadl** will map to their respective security services identities on every machine in the cluster. Refer to *IBM Reliable Scalable Cluster Technology for AIX: Technical Reference*, SA22-7800 for a description of the syntax used in the **ctsec_map.global** file.

5. Create UNIX groups. The LoadLeveler **administrator** group and **services** group need to be created for every machine in the cluster and should contain the local identities of members. This can be done either by using SMIT or the **mkgroup** command.

   For example, to create the group lladmin which lists the LoadLeveler administrators:

   ```
   mkgroup "users=sam,betty,loadl" lladmin
   ```

   These groups must be created on each machine in the LoadLeveler cluster and must contain the same entries.

   To create the group llsvcs which lists the identity under which LoadLeveler daemons run using the default id of loadl:

   ```
   mkgroup users=loadl llsvcs
   ```

   These groups must be created on each machine in the LoadLeveler cluster and must contain the same entries.

6. Add or update these keywords in the LoadLeveler configuration:

   ```
   SEC_ENABLEMENT=CTSEC
   SEC_ADMIN_GROUP=name of lladmin group
   SEC_SERVICES_GROUP=group name that contains identities of LoadLeveler daemons
   ```

   The **SEC_ENABLEMENT=CTSEC** keyword indicates that CtSec services mechanism should be used. **SEC_ADMIN_GROUP** points to the name of the UNIX group which contains the local identities of the LoadLeveler administrators. The **SEC_SERVICES_GROUP** keyword points to the name of the UNIX group which contains the local identity of the LoadLeveler daemons. All LoadLeveler daemons run as the LoadLeveler user ID. Refer to step 5 for discussion of the **administrators** and **services** groups.

7. Update the **.rhosts** file in each user's home directory. This file is used to identify which UNIX identities can run LoadLeveler jobs on the local host machine. If the file does not exist in a user's home directory, you must create it. The **.rhosts** file must contain entries which specify all host and user

combinations allowed to submit jobs which will run as the local user, either explicitly or through the use of wildcards.

Entries in the **.rhosts** file are specified this way:

```
HostNameField [UserNameField]
```

Refer to *IBM AIX Files Reference*, SC23-4168 for further details about the **.rhosts** file format.

Tips for configuring LoadLeveler to use CtSec services: When using CtSec services for LoadLeveler, each machine in the LoadLeveler cluster must be set up properly.

CtSec authenticates network identities based on trust established between individual machines in a cluster, based on local host configurations. Because of this it is possible for most of the cluster to run correctly but to have transactions from certain machines experience authentication or authorization problems.

If unexpected authentication or authorization problems occur in a LoadLeveler cluster with CtSec enabled, check that the steps in "Steps for enabling CtSec services" on page 62 were correctly followed for each machine in the LoadLeveler cluster.

If any machine in a LoadLeveler cluster is improperly configured to run CtSec you may see that:

- Users cannot perform user tasks (such as cancel) for jobs they submitted.

  Either the machine the job was submitted from or the machine the user operation was submitted from (or both) do not contain mapping files for the user that specify the same security services identity. The user should attempt the operation from the same machine the job was submitted from and record the results. If the user still cannot perform a user task on a job they submitted, then they should contact the LoadLeveler administrator who should review the steps in "Steps for enabling CtSec services" on page 62.
- LoadLeveler daemons fail to communicate.

  When LoadLeveler daemons communicate they must first authenticate each other. If the daemons cannot authenticate a message will be put in the daemon log indicating an authentication failure. Ensure the Trusted Hosts List on all LoadLeveler nodes contains the correct entries for all of the nodes in the LoadLeveler cluster. Also, make sure that the LoadLeveler Services group on all nodes of the LoadLeveler cluster contains the local identity for the LoadLeveler user name. The **ctsec_map.global** must contain mapping rules to map the LoadLeveler user name from every machine in the LoadLeveler cluster to the local identity for the LoadLeveler user name. An example of what may happen when daemons fail to communicate is that an alternate central manager may take over while the primary central manager is still active. This can occur when the alternate central manager does not trust the primary central manager.

Limiting which security mechanisms LoadLeveler can use: As more security mechanisms become available, they must be configured for LoadLeveler's use. If there are security mechanisms configured for CtSec that are not configured for LoadLeveler's use, then the LoadLeveler configuration keyword **SEC_IMPOSED_MECHS** must specify the mechanisms configured for LoadLeveler.

Defining usage policies for consumable resources

The LoadLeveler scheduler can schedule jobs based on the availability of consumable resources.

You can use the following keywords to configure consumable resources:

- **ENFORCE_RESOURCE_MEMORY**
- **ENFORCE_RESOURCE_POLICY**
- **ENFORCE_RESOURCE_SUBMISSION**
- **ENFORCE_RESOURCE_USAGE**
- **FLOATING_RESOURCES**
- **RESOURCES**
- **SCHEDULE_BY_RESOURCES**

For information about configuration keyword syntax and other details, see Chapter 10, "Configuration keyword reference," on page 231.

Gathering job accounting data

Your organization may have a policy of charging users or groups of users for the amount of resources that their jobs consume.

You can do this using LoadLeveler's accounting feature. Using this feature, you can produce accounting reports that contain job resource information for completed serial and parallel job steps. You can also view job resource information on jobs that are continuing to run.

The accounting record for a job step will contain separate sets of resource usage data for each time a job step is dispatched to run. For example, the accounting record for a job step that is vacated and then started again will contain two sets of resource usage data. The first set of resource usage data is for the time period when the job step was initially dispatched until the job step was vacated. The second set of resource usage data is for the time period for when the job step is dispatched after the vacate until the job step completes.

The job step's accounting data that is provided in the **llsummary** short listing and in the user mail will contain only one set of resource usage data. That data will be from the last time the job step was dispatched to run. For example, the mail message for job step completion for a job step that is checkpointed with the hold (**-h**) option and then restarted, will contain the set of resource usage data only for the dispatch that restarted the job from the checkpoint. To obtain the resource usage data for the entire job step, use the detailed **llsummary** command or accounting API.

The following keywords allow you to control accounting functions:

- **ACCT**
- **ACCT_VALIDATION**
- **GLOBAL_HISTORY**
- **HISTORY_PERMISSION**
- **JOB_ACCT_Q_POLICY**
- **JOB_LIMIT_POLICY**

For example, the following section of the configuration file specifies that the accounting function is turned on. It also identifies the default module used to perform account validation and the directory containing the global history files:

```
ACCT                = A_ON A_VALIDATE
ACCT_VALIDATION     = $(BIN)/llacctval
GLOBAL_HISTORY      = $(SPOOL)
```

Table 13 lists the topics related to configuring, gathering and using job accounting data.

Table 13. Roadmap of tasks for gathering job accounting data

To learn about:	Read the following:
Configuring LoadLeveler to gather job accounting data	• "Collecting job resource data on serial and parallel jobs" • "Collecting job resource data based on machines" on page 68 • "Collecting job resource data based on events" on page 68 • "Collecting job resource information based on user accounts" on page 69 • "Collecting accounting data for reservations" on page 67 • "Collecting the accounting information and storing it into files" on page 69 • "64-bit support for accounting functions" on page 71 • "Example: Setting up job accounting files" on page 71
Managing accounting data	• "Producing accounting reports" on page 70 • "Correlating AIX and LoadLeveler accounting records" on page 70 • **llacctmrg** in *LoadLeveler: Command and API Reference* • **llsummary** in *LoadLeveler: Command and API Reference*
Correctly specifying configuration keywords	Chapter 10, "Configuration keyword reference," on page 231

Collecting job resource data on serial and parallel jobs

Information on completed serial and parallel job steps is gathered using the UNIX *wait3* system call. Information on non-completed serial and parallel jobs is gathered in a platform-dependent manner by examining data from the UNIX process.

Accounting information on a completed serial job step is determined by accumulating resources consumed by that job on the machines that ran the job. Similarly, accounting information on completed parallel job steps is gathered by accumulating resources used on all of the nodes that ran the job step.

You can also view resource consumption information on serial and parallel jobs that are still running by specifying the **-x** option of the **llq** command. To enable **llq -x**, specify the following keywords in the configuration file:

- **ACCT** = A_ON A_DETAIL
- **JOB_ACCT_Q_POLICY** = *number*

Collecting accounting information for recurring jobs

For recurring jobs, accounting records are written as each occurrence of each step of the job completes. The reservation ID field in the accounting record can be used to distinguish one occurrence from another.

Collecting accounting data for reservations

LoadLeveler can collect accounting data for reservations, which are set periods of time during which node resources are reserved for the use of particular users or groups.

To enable recording of reservation information, specify the following keywords in the configuration:

- To turn on accounting for reservations, add the **A_RES** flag to the **ACCT** keyword.
- To specify a file other than the default history file to contain the data, use the **RESERVATION_HISTORY** keyword.

See Chapter 10, "Configuration keyword reference," on page 231 for details about the **ACCT** and **RESERVATION_HISTORY** keywords.

When these keyword values are set and a reservation ends or is canceled, LoadLeveler records the following information:

- The reservation ID
- The time at which the reservation was created
- The user ID of the reservation owner
- The name of the owning group
- Requested and actual start times
- Requested and actual duration
- Actual time at which the reservation ended or was canceled
- The name of the flexible job ID for a flexible reservation
- Whether the reservation was created with the `SHARED` or `REMOVE_ON_IDLE` options
- A list of users and a list of groups that were authorized to use the reservation
- The number of reserved nodes
- The names of reserved nodes
- The name of the notification program
- The list of notification program arguments
- The list of floating resources
- The number of reserved Blue Gene compute nodes
- The connectivity of the Blue Gene block
- The shape of the Blue Gene block
- The number of midplanes used by the Blue Gene block
- The midplanes used by the Blue Gene block
- The name of the Blue Gene block

This reservation information is appended in a single line to the reservation history file for the reservation. The format of reservation history data is:

```
Reservation ID!Reservation Type!Reservation Creation Time!Owner!Owning Group!Start Time! \
 Actual Start Time!Duration!Actual Duration!Actual End Time!Flexible Job ID! SHARED(yes|no)! \
 Number of Nodes!Nodes!Floating Resources!Number of BG Compute Nodes!BG Connectivity!BG Shape! \
 Number of BG Midplanes!BG Midplanes!BG Block Name
```

In reservation history data:

- The unit of measure for start times and end times is the number of seconds since January 1, 1970.
- The unit of time for durations is seconds.

Note: As each occurrence of a recurring reservation completes, an accounting record is appended to the reservation history file. The format of the record is identical to that of a one time reservation. In the record, the Reservation ID includes the occurrence ID of the completed reservation.

When you cancel the *entire* recurring reservation (as opposed to only one occurrence being canceled), one additional accounting record is written. This record is based on the state of the reservation:

- If an occurrence is ACTIVE, then the end time and duration of that occurrence is set and an accounting record written.
- If there are not any ACTIVE occurrences, then an accounting record will be written for the next scheduled occurrence. This is similar to the accounting record that is written when you cancel a one time reservation in the WAITING state.

The following is an example of a reservation history file entry:

```
cnbladel1n04v7.clusters.com.23.r!ONE_TIME!1303805213!ytxiang!No_Group!1303805400! \
 1303805400!120!120!1303805520!!no!no!!!/u/ytxiang/rpm/1.cmd!!1!cnbladel1n04v8!abc(2)!
cnbladel1n04v7.clusters.com.24.r.0!RECURRING!1303805239!ytxiang!No_Group!1303805400! \
 1303805400!180!180!1303805580!!no!no!!!/u/ytxiang/rpm/2.cmd!!1!cnbladel1n04v7!abc(3)! \
cnbladel1n04v7.clusters.com.25.r!FLEXIBLE!1303805262!ytxiang!No_Group!1303805580! \
 1303805580!240!240!1303805820!cnbladel1n04v7.clusters.com.36.0!no!no!!!/u/ytxiang/rpm/ \
 1.cmd!!2!cnbladel1n04v8,cnbladel1n04v7!abc(5)!
```

To collect the reservation information stored in the history file, use the **llacctmrg** command with the **-R** option. For **llacctmrg** command syntax, see *LoadLeveler: Command and API Reference*.

To format reservation history data contained in a file, use the sample script `llreshist.pl` in directory `${RELEASEDIR}/samples/llres/`.

Collecting job resource data based on machines

LoadLeveler can collect job resource usage information for every machine on which a job may run. A job may run on more than one machine because it is a parallel job or because the job is vacated from one machine and rescheduled to another machine.

To enable recording of resources by machine, you need to specify **ACCT = A_ON A_DETAIL** in the configuration.

The machine's speed is part of the data collected. With this information, an installation can develop a charge back program which can charge more or less for resources consumed by a job on different machines. For more information on a machine's speed, refer to the machine stanza information. See “Defining machines” on page 89.

Collecting job resource data based on events

In addition to collecting job resource information based upon machines used, you can gather this information based upon an event or time that you specify. For example, you may want to collect accounting information at the end of every work shift or at the end of every week or month. To collect accounting information on all machines in this manner, use the **llctl** command with the **capture** parameter:

llctl -g capture *eventname*

eventname is any string of continuous characters (no white space is allowed) that defines the event about which you are collecting accounting data. For example, if you were collecting accounting data on the *graveyard* work shift, your command could be:

```
llctl -g capture graveyard
```

This command allows you to obtain a snapshot of the resources consumed by active jobs up to and including the moment when you issued the command. If you

want to capture this type of information on a regular basis, you can set up a crontab entry to invoke this command regularly. For example:

```
#  sample crontab for accounting
#  shift crontab 94/8/5
#
# Set up three shifts, first, second, and graveyard shift.
#  Crontab entries indicate the end of shift.
#
#M  H d m day command
#
00 08 * *  * /u/loadl/bin/llctl -g capture graveyard
00 16 * *  * /u/loadl/bin/llctl -g capture first
00 00 * *  * /u/loadl/bin/llctl -g capture second
```

For more information on the **llctl** command, refer to *LoadLeveler: Command and API Reference*. For more information on the collection of accounting records, see the **llq** command in *LoadLeveler: Command and API Reference*.

Collecting job resource information based on user accounts

If your installation is interested in keeping track of resources used on an account basis, you can require all users to specify an account number in their job command files. They can specify this account number with the **account_no** keyword which is explained in detail in "Job command file keyword descriptions" on page 335. Interactive POE jobs can specify an account number using the **LOADL_ACCOUNT_NO** environment variable.

LoadLeveler validates this account number by comparing it against a list of account numbers specified for the user in the user stanza in the administration file or the user table in the configuration database.

Account validation is under the control of the **ACCT** keyword in the configuration file or database. The routine that performs the validation is called **llacctval**. You can supply your own validation routine by specifying the **ACCT_VALIDATION** keyword in the configuration file or database. The following are passed as character string arguments to the validation routine:

- User name
- User's login group name
- Account number specified on the Job
- Blank-separated list of account numbers obtained from the user's stanza in the administration file or database.

The account validation routine must exit with a return code of zero if the validation succeeds. If it fails, the return code is a nonzero number.

Collecting the accounting information and storing it into files

LoadLeveler stores the accounting information that it collects in a file called *history* in the spool directory of the machine that initially scheduled this job, the Schedd machine. Data on parallel jobs are also stored in the *history* files.

Resource information collected on the LoadLeveler job is constrained by the capabilities of the wait3 system call. Information for processes which fork child processes will include data for those child processes as long as the parent process waits for the child process to terminate. Complete data may not be collected for jobs which are not composed of simple parent/child processes. For example, if you have a LoadLeveler job which invokes an rsh command to execute a function on another machine, the resources consumed on the other machine will not be collected as part of the LoadLeveler accounting data.

LoadLeveler accounting uses the following types of files:

- The local history file which is local to each Schedd machine is where job resource information is first recorded. These files are usually named *history* and are located in the spool directory of each Schedd machine, but you may specify an alternate name with the **HISTORY** keyword in either the global or local configuration file or the configuration database.
- The global history file is a combination of the history files from some or all of the machines in the LoadLeveler cluster merged together. The command **llacctmrg** is used to collect files together into a global file. As the files are collected from each machine, the local history file for that machine is reset to contain no data. The file is named *globalhist.YYYYMMDDHHmm*. You may specify the directory in which to place the file when you invoke the **llacctmrg** command or you can specify the directory with the **GLOBAL_HISTORY** keyword in the configuration file or database. The default value set up in the sample configuration file is the local spool directory.

Producing accounting reports

You can produce three types of accounting reports called the *short*, *long*, and *extended* versions. As their names imply, the short version of the report is a brief listing of the resources used by LoadLeveler jobs. The long version provides more comprehensive detail with summarized resource usage, and the extended version of the report provides the comprehensive detail with detailed resource usage.

If you do not specify a report type, you will receive the default short version. The short report displays the number of jobs along with the total CPU usage according to user, class, group, and account number. The extended version of the report displays all of the data collected for every job.

- For examples of the short and extended versions of the report, see the **llsummary** command in *LoadLeveler: Command and API Reference*.
- For information on the accounting APIs, see the "Application programming interfaces (APIs)" topic in *LoadLeveler: Command and API Reference*.

Correlating AIX and LoadLeveler accounting records

For jobs running on AIX systems, you can use a job accounting key to correlate AIX accounting records with LoadLeveler accounting records. The job accounting key uniquely identifies each job step. LoadLeveler derives this key from the job key and the date and time at which the job entered the queue (see the **QDate** variable description in “LoadLeveler variables” on page 286). The key is associated with the starter process for the job step and any of its child processes.

For checkpointed jobs, LoadLeveler does not change the job accounting key, regardless of how it restarts the job step. Jobs restarted from a checkpoint file or through a new job step retain the job accounting key for the original job step.

To access the job accounting key for a job step, you can use the following interfaces:

- The **llsummary** command, requesting the long version of the report. For details about using this command, see *LoadLeveler: Command and API Reference*.
- The **GetHistory** subroutine. For details about using this subroutine, see *LoadLeveler: Command and API Reference*.
- The **ll_get_data** subroutine, through the **LL_StepAcctKey** specification. For details about using this subroutine, see *LoadLeveler: Command and API Reference*.

For information about AIX accounting records, see the system accounting topic in *AIX System Management Guide: Operating System and Devices*.

64-bit support for accounting functions

LoadLeveler 64-bit support for accounting functions includes:

- Statistics of jobs such as usage, limits, consumable resources, and other 64-bit integer data are preserved in the history file as rusage64, rlimit64 structures and as data items of type int64_t.
- The **LL_job_step** structure defined in llapi.h allows access to the 64-bit data items either as data of type int64_t or as data of type int32_t. In the latter case, the returned values may be truncated.
- The **llsummary** command displays 64-bit information where appropriate.
- The data access API supports both 64-bit and 32-bit access to accounting and usage information in a history file. See the examples of using the data access API in *LoadLeveler: Command and API Reference* for an example of how to use the **ll_get_data** subroutine to access information stored in a LoadLeveler history file.

Example: Setting up job accounting files

You can perform all of the steps included in this sample procedure or just the ones that apply to your situation. The sample procedure shown in Table 14 walks you through the process of collecting account data.

1. Edit the configuration keywords according to the following table:

Table 14. Collecting account data - modifying configuration keywords

Edit this keyword:	To:
ACCT	Turn accounting and account validation on and off and specify detailed accounting.
ACCT_VALIDATION	Specify the account validation routine.
GLOBAL_HISTORY	Specify a directory in which to place the global history files.

2. Specify account numbers and set up account validation by performing the following steps:
   a. Specify a list of account numbers a user may use when submitting jobs, by using the **account** keyword in the user stanza in the administration file.
   b. Instruct users to associate an account number with their job, by using the **account_no** keyword in the job command file.
   c. Specify the **ACCT_VALIDATION** configuration keyword that identifies the module that will be called to perform account validation. The default module is called **llacctval**. You can replace this module with your installation's own accounting routine by specifying a new module with this keyword.
3. Specify machines and their weights by using the **speed** keyword in a machine's machine stanza in the administration file.

   Also, if you have in your cluster machines of differing speeds and you want LoadLeveler accounting information to be normalized for these differences, specify **cpu_speed_scale=true** in each machine's respective machine stanza.

   For example, suppose you have a cluster of two machines, called A and B, where Machine B is three times as fast as Machine A. Machine A has **speed=1.0**, and Machine B has **speed=3.0**. Suppose a job runs for 12 CPU seconds on Machine A. The same job runs for 4 CPU seconds on Machine B. When you specify **cpu_speed_scale=true**, the accounting information collected on Machine B for that job shows the normalized value of 12 CPU seconds rather than the actual 4 CPU seconds.
4. Merge multiple files collected from each machine into one file, using the **llacctmrg** command.

5. Report job information on all the jobs in the history file, using the **llsummary** command.

Managing job status through control expressions

You can control running jobs by using five control functions as Boolean expressions in the configuration.

These functions are useful primarily for serial jobs. You define the expressions, using normal C conventions, with the following functions:
- START
- SUSPEND
- CONTINUE
- VACATE
- KILL

The expressions are evaluated for each job running on a machine using both the job and machine attributes. Some jobs running on a machine may be suspended while others are allowed to continue.

The START expression is evaluated twice; once to see if the machine can accept jobs to run and second to see if the specific job can be run on the machine. The other expressions are evaluated after the jobs have been dispatched and in some cases, already running.

When evaluating the START expression to determine if the machine can accept jobs, **Class != "Z"** evaluates to true only if `Z` is not in the class definition. This means that if two different classes are defined on a machine, **Class != "Z"** (where `Z` is one of the defined classes) always evaluates to false when specified in the START expression and, therefore, the machine will not be considered to start jobs.

Typically, machine load average, keyboard activity, time intervals, and job class are used within these various expressions to dynamically control job execution.

For additional information about:
- Time-related variables that you may use for this keyword, see "Variables to use for setting times" on page 291.
- Coding these control expressions in the configuration keywords, see Chapter 10, "Configuration keyword reference," on page 231.

How control expressions affect jobs

After LoadLeveler selects a job for execution, the job can be in any of several states. Figure 10 on page 73 shows how the control expressions can affect the state a job is in. The rectangles represent job or daemon states (Idle, Completed, Running, Suspended, and Vacating) and the diamonds represent the control expressions (Start, Suspend, Continue, Vacate, and Kill).

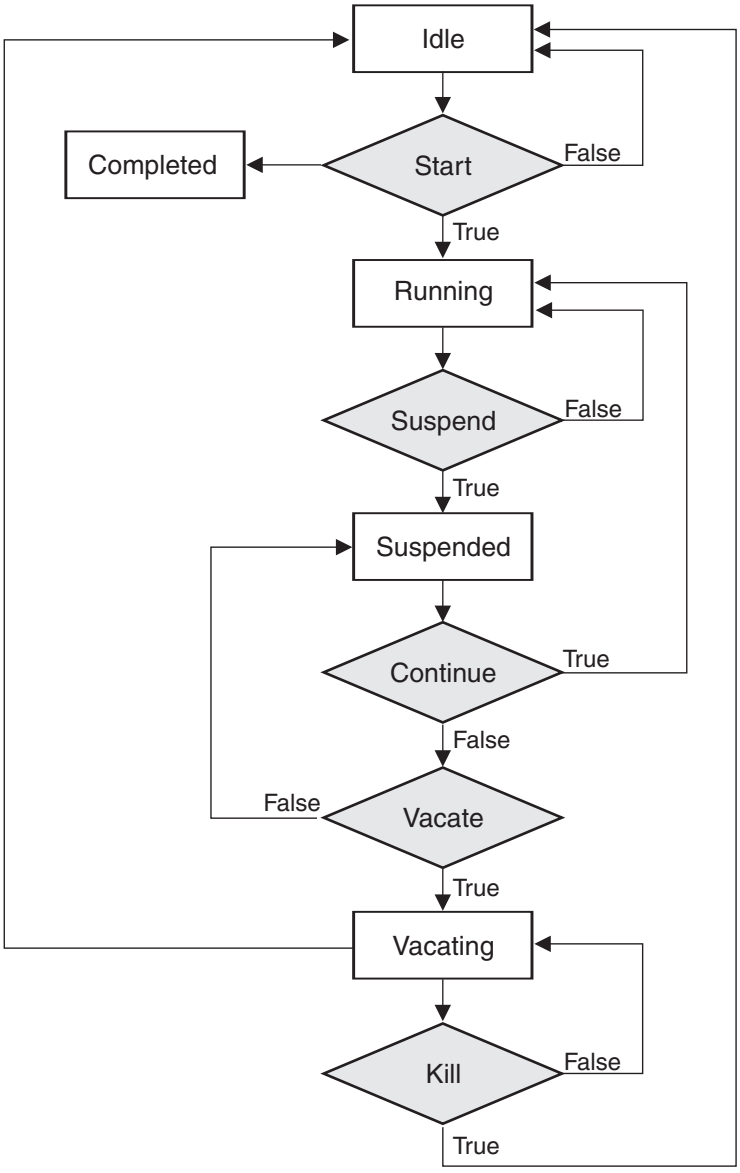

Figure 10. How control expressions affect jobs

Criteria used to determine when a LoadLeveler job will enter Start, Suspend, Continue, Vacate, and Kill states are defined in the LoadLeveler configuration and they can be different for each machine in the cluster. They can be modified to meet local requirements.

Tracking job processes

When a job terminates, its orphaned processes may continue to consume or hold resources, thereby degrading system performance, or causing jobs to hang or fail.

Process tracking allows LoadLeveler to cancel any processes (throughout the entire cluster), left behind when a job terminates. Process tracking is required to do preemption by the suspend method when running either the BACKFILL or API schedulers. Process tracking is optional in all other cases.

When process tracking is enabled, all child processes are terminated when the main process terminates. This will include any background or orphaned processes started in the prolog, epilog, user prolog, and user epilog.

Process tracking on LoadLeveler for Linux is supported only on RHEL 6.

There are two keywords used in specifying process tracking:

PROCESS_TRACKING
To activate process tracking, set **PROCESS_TRACKING=TRUE** in the LoadLeveler global configuration file or database. By default, **PROCESS_TRACKING** is set to **FALSE**.

PROCESS_TRACKING_EXTENSION
This keyword is for AIX only. This keyword specifies the path to the loadable kernel module **LoadL_pt_ke** in the local or global configuration file or database. If the **PROCESS_TRACKING_EXTENSION** keyword is not supplied, then LoadLeveler will search the **$HOME/bin** default directory.

The process tracking kernel extension is not unloaded when the **startd** daemon terminates on systems running AIX. Therefore, if a mismatch in the version of the loaded kernel extension and the installed kernel extension is found when the **startd** starts up the daemon will exit. In this case, a reboot of the node is needed to unload the currently loaded kernel extension. If you install a new version of the LoadLeveler resource manager, which contains a new version of the kernel extension, you may need to reboot the node.

For information about configuration keyword syntax and other details, see Chapter 10, “Configuration keyword reference,” on page 231.

Querying multiple LoadLeveler clusters

This topic applies only to those installations having more than one LoadLeveler cluster, where the separate clusters have not been organized into a multicluster environment.

To organize separate LoadLeveler clusters into a multicluster environment, see “LoadLeveler multicluster support” on page 149.

You can query, submit, or cancel jobs in multiple LoadLeveler clusters by setting up a master configuration file for each cluster and using the **LOADL_CONFIG** environment variable to specify the name of the master configuration file that the LoadLeveler commands must use. The master configuration file must be located in the **/etc** directory and the file name must have a format of *base_name***.cfg** where *base_name* is a user defined identifier for the cluster.

The default name for the master configuration file is **/etc/LoadL.cfg**. The format for the **LOADL_CONFIG** environment variable is **LOADL_CONFIG=/etc/** *base_name***.cfg** or **LOADL_CONFIG=***base_name*. When you use the form **LOADL_CONFIG=***base_name*, the prefix **/etc** and suffix **.cfg** are appended to the *base_name*.

The following example explains how you can set up a machine to query multiple clusters:

You can configure **/etc/LoadL.cfg** to point to the configuration files for the "default" cluster, and you can configure **/etc/othercluster.cfg** to point to the configuration files of another cluster which the user can select.

For example, you can enter the following query command:

```
$ llq
```

The **llq** command uses the configuration from **/etc/LoadL.cfg** and queries job information from the "default" cluster.

To send a query to the cluster defined in the configuration file of **/etc/othercluster.cfg**, enter:

```
$ env LOADL_CONFIG=othercluster llq
```

Note that the machine from which you issue the **llq** command is considered as a submit-only machine by the other cluster.

Handling switch-table errors

Configuration keywords can be used to control how LoadLeveler responds to switch-table errors.

You may use the following configuration keywords to control how LoadLeveler responds to switch-table errors:

- ACTION_ON_SWITCH_TABLE_ERROR
- DRAIN_ON_SWITCH_TABLE_ERROR
- RESUME_ON_SWITCH_TABLE_ERROR_CLEAR

For information about configuration keyword syntax and other details, see Chapter 10, "Configuration keyword reference," on page 231.

Providing additional job-processing controls through installation exits

LoadLeveler allows administrators to further configure the environment through installation exits.

Table 15 lists these additional job-processing controls.

Table 15. Roadmap of administrator tasks accomplished through installation exits

To learn about:	Read the following:
Writing a program to control when jobs are scheduled to run	"Controlling the central manager scheduling cycle" on page 76
Writing a pair of programs to override the default LoadLeveler DCE authentication method	"Handling DCE security credentials" on page 77
Writing a program to refresh an AFS token when a job starts	"Handling an AFS token" on page 78
Writing a program to check or modify job requests when they are submitted	"Filtering a job script" on page 79
Writing programs to run before and after job requests	"Writing prolog and epilog programs" on page 80
Overriding the LoadLeveler default mail notification method	"Using your own mail program" on page 85
Defining a cluster metric to determine where a remote job is distributed	See the **CLUSTER_METRIC** configuration keyword description in Chapter 10, "Configuration keyword reference," on page 231.

Table 15. Roadmap of administrator tasks accomplished through installation exits (continued)

To learn about:	Read the following:
Defining cluster user mapper for multicluster environment	See the **CLUSTER_USER_MAPPER** configuration keyword description in Chapter 10, "Configuration keyword reference," on page 231.
Correctly specifying configuration keywords	Chapter 10, "Configuration keyword reference," on page 231
Writing an installation exit that can determine the frequency to use to run a job	"Determining the frequency to use to run a job" on page 85

Controlling the central manager scheduling cycle

To determine when to run the LoadLeveler scheduling algorithm, the central manager uses the values set in the configuration for the **NEGOTIATOR_INTERVAL** and the **NEGOTIATOR_CYCLE_DELAY** keywords. The central manager will run the scheduling algorithm every **NEGOTIATOR_INTERVAL** seconds, unless some event takes place such as the completion of a job or the addition of a machine to the cluster. In such cases, the scheduling algorithm is run immediately. When **NEGOTIATOR_CYCLE_DELAY** is set, a minimum of **NEGOTIATOR_CYCLE_DELAY** seconds will pass between the central manager's scheduling attempts, regardless of what other events might take place. When the **NEGOTIATOR_INTERVAL** is set to zero, the central manager will not run the scheduling algorithm until instructed to do so by an authorized process. This setting enables your program to control the central manager's scheduling activity through one of the following:

- The **llrunscheduler** command.
- The **ll_run_scheduler** subroutine.

Both the command and the subroutine instruct the central manager to run the scheduling algorithm.

You might choose to use this setting if, for example, you want to write a program that directly controls the assignment of the system priority for all LoadLeveler jobs. In this particular case, you would complete the following steps to control system priority assignment and the scheduling cycle:

1. Decide the following:
   - Which system priority value to assign to jobs from specific sources or with specific resource requirements.
   - How often the central manager should run the scheduling algorithm. Your program has to be designed to issue the **ll_run_scheduler** subroutine at regular intervals; otherwise, LoadLeveler will not attempt to schedule any job steps.

   You also need to understand how changing the system priority affects the job queue. After you successfully use the **ll_modify** subroutine or the **llmodify** command to change system priority values, LoadLeveler will not readjust the values for those job steps when the negotiator recalculates priorities at regular intervals set through the **NEGOTIATOR_RECALCULATE_SYSPRIO_INTERVAL** keyword. Also, you can change the system priority for jobs only when those jobs are in the Idle state or a state similar to it. To determine which job states are similar to the Idle state or to the Running state, see the table in "LoadLeveler job states" on page 19.
2. Code a program to use LoadLeveler APIs to perform the following functions:
   a. Use the Data Access APIs to obtain data about all jobs.

b. Determine whether jobs have been added or removed.
c. Use the **ll_modify** subroutine to set the system priority for the LoadLeveler jobs. The values you set through this subroutine will not be readjusted when the negotiator recalculates job step priorities.
d. Use the **ll_run_scheduler** subroutine to instruct the central manager to run the scheduling algorithm.
e. Set a timer for the scheduling interval, to repeat the scheduling instruction at regular intervals. This step is required to replace the effect of setting the configuration keyword **NEGOTIATOR_CYCLE_DELAY**, which LoadLeveler ignores when **NEGOTIATOR_INTERVAL** is set to zero.

3. In the configuration file or database, set values for the following keywords:
   - Set the **NEGOTIATOR_INTERVAL** keyword to zero to stop the central manager from automatically recalculating system priorities for jobs.
   - (Optional) Set the **SYSPRIO_THRESHOLD_TO_IGNORE_STEP** keyword to specify a threshold value. If the system priority assigned to a job step is less than this threshold value, the job will remain idle.
4. Issue the **llctl** command with either the **reconfig** or **recycle** keyword. Otherwise, LoadLeveler will not process the modifications you made to the configuration file or database.
5. (Optional) To make sure that the central manager's automatic scheduling activity has been disabled (by setting the **NEGOTIATOR_INTERVAL** keyword to zero), use the **llstatus** command.
6. Run your program under a user ID with administrator authority.

Once this procedure is complete, you might want to use one or more of the following commands to make sure that jobs are scheduled according to the correct system priority. The value of *q_sysprio* in command output indicates the system priority for the job step.

- Use the command **llq -s** to detect whether a job step is idle because its system priority is below the value set for the **SYSPRIO_THRESHOLD_TO_IGNORE_STEP** keyword.
- Use the command **llq -l** to display the previous system priority for a job step.
- When unusual circumstances require you to change system priorities manually:
  1. Use the command **llmodify -s** to set the system priority for LoadLeveler jobs. The values you set through this command will not be readjusted when the negotiator recalculates job step priorities.
  2. Use the **llrunscheduler** command to instruct the central manager to run the scheduling algorithm.

Handling DCE security credentials

You can write a pair of programs to override the default LoadLeveler DCE authentication method. To enable the programs, use the **DCE_AUTHENTICATION_PAIR** keyword in your configuration file or database:

- As an alternative, you can also specify the program pair:

```
DCE_AUTHENTICATION_PAIR = $(BIN)/llgetdce, $(BIN)/llsetdce
```

Specifying the **DCE_AUTHENTICATION_PAIR** keyword enables LoadLeveler support for forwarding DCE credentials to LoadLeveler jobs. You may override the default function provided by LoadLeveler to establish DCE credentials by substituting your own programs.

Using the alternative program pair: llgetdce and llsetdce: The program pair, **llgetdce** and **llsetdce**, forwards DCE credentials by copying credential cache files

from the submitting machine to the executing machines. While this technique may require less overhead, it has been known to produce credentials on the executing machines which are not fully capable of being forwarded by rsh commands. This is the only pair of programs offered in earlier releases of LoadLeveler.

Forwarding DCE credentials: An example of a credentials object is a character string containing the DCE principle name and a password.

program1 writes the following to standard output:

- The length of the handle to follow
- The handle

If *program1* encounters errors, it writes error messages to standard error.

program2 receives the following as standard input:

- The length of the handle to follow
- The same handle written by *program1*

program2 writes the following to standard output:

- The length of the login context to follow
- An exportable DCE login context, which is the idl_byte array produced from the sec_login_export_context DCE API call. For more information, see the DCE Security Services API chapter in the *Distributed Computing Environment for AIX: Application Development Reference*.
- A character string suitable for assigning to the KRB5CCNAME environment variable This string represents the location of the credentials cache established in order for *program2* to export the DCE login context.

If *program2* encounters errors, it writes error messages to standard error. The parent process, the LoadLeveler starter process, writes those messages to the starter log.

For examples of programs that enable DCE security credentials, see the **samples/lldce** subdirectory in the release directory.

Handling an AFS token

You can write a program, run by the scheduler, to refresh an AFS token when a job is started. To invoke the program, use the **AFS_GETNEWTOKEN** keyword in your configuration file.

Before running the program, LoadLeveler sets up standard input and standard output as pipes between the program and LoadLeveler. LoadLeveler also sets up the following environment variables:

LOADL_STEP_OWNER
: The owner (UNIX user name) of the job

LOADL_STEP_COMMAND
: The name of the command the user's job step invokes.

LOADL_STEP_CLASS
: The class this job step will run.

LOADL_STEP_ID
: The step identifier, generated by LoadLeveler.

LOADL_JOB_CPU_LIMIT
: The number of CPU seconds the job is limited to.

LOADL_WALL_LIMIT
: The number of wall clock seconds the job is limited to.

LoadLeveler writes the following current AFS credentials, in order, over the standard input pipe:
- The **ktc_principal** structure indicating the service.
- The **ktc_principal** structure indicating the client.
- The **ktc_token** structure containing the credentials.

The ktc_principal structure is defined in the AFS header file **afs_rxkad.h**. The ktc_token structure is defined in the AFS header file **afs_auth.h**.

LoadLeveler expects to read these same structures in the same order from the standard output pipe, except these should be refreshed credentials produced by the installation exit.

The installation exit can modify the passed credentials (to extend their lifetime) and pass them back, or it can obtain new credentials. LoadLeveler takes whatever is returned and uses it to authenticate the user prior to starting the user's job.

Filtering a job script

You can write a program to filter a job script when the job is submitted to the local cluster and when the job is submitted from a remote cluster. This program can, for example, modify defaults or perform site specific verification of parameters. To invoke the local job filter, specify the **SUBMIT_FILTER** keyword in your configuration file or database. To invoke the remote job filter, specify the **CLUSTER_REMOTE_JOB_FILTER** keyword in your configuration file or database. For more information on these keywords, see the **SUBMIT_FILTER** or **CLUSTER_REMOTE_JOB_FILTER** keyword in Chapter 10, "Configuration keyword reference," on page 231.

LoadLeveler sets the following environment variables when the program is invoked:

LOADL_ACTIVE
: LoadLeveler version

LOADL_STEP_COMMAND
: Job command file name

LOADL_STEP_ID
: The job identifier, generated by LoadLeveler

LOADL_STEP_OWNER
: The owner (UNIX user name) of the job

For details about specific keyword syntax and use in the configuration, see Chapter 10, "Configuration keyword reference," on page 231.

A sample filter script, to add island scheduling requirements to all jobs, can be found in the samples directory of the LoadLeveler installation.

On AIX:

```
/usr/lpp/LoadL/scheduler/full/samples/submit_filter.pl
/usr/lpp/LoadL/resmgr/full/samples/submit_filter.pl
```

On Linux:

```
/opt/ibmll/LoadL/scheduler/full/samples/submit_filter.pl
/opt/ibmll/LoadL/resmgr/full/samples/submit_filter.pl
```

The **submit_filter.pl** filter will have the following characteristics:
- All job steps will end up with **node_usage=not_shared**.
- All job steps will end up with **node_topology=island**.

- The script will differentiate between batch and interactive job steps based on class name. For interactive job steps it will add a preference for **island == "island01"**.

Writing prolog and epilog programs

An administrator can write *prolog* and *epilog* installation exits that can run before and after a LoadLeveler job runs, respectively.

Prolog and epilog programs fall into two types:

- Those that run as the LoadLeveler user ID.
- Those that run in a user's environment.

Depending on the type of processing you want to perform before or after a job runs, specify one or more of the following configuration keywords, in any combination:

- To run a prolog or epilog program under the LoadLeveler user ID, specify `JOB_PROLOG` or `JOB_EPILOG`, respectively.
- To run a prolog or epilog program under the user's environment, specify `JOB_USER_PROLOG` or `JOB_USER_EPILOG`, respectively.

You do not have to provide a prolog/epilog pair of programs. You may, for example, use only a prolog program that runs under the LoadLeveler user ID.

For details about specific keyword syntax and use in the configuration, see Chapter 10, "Configuration keyword reference," on page 231.

Note: If process tracking is enabled and your prolog or epilog program invokes the **mailx** command, set the **sendwait** variable to prevent the background mail process from being killed by process tracking.

A user environment prolog or epilog runs with AFS authentication if installed and enabled. For security reasons, you must code these programs on the machines where the job runs *and* on the machine that schedules the job. If you do not define a value for these keywords, the user environment prolog and epilog settings on the executing machine are ignored.

The user environment prolog can set environment variables for the job by sending information to standard output in the following format:

```
env id = value
```

Where:

id Is the name of the environment variable
value Is the value (setting) of the environment variable

Note: Each line of output can contain a maximum of 65,534 characters. All lines containing more than 65,534 characters will be ignored.

For example, the user environment prolog sets the environment variable **STAGE_HOST** for the job:

```
#!/bin/sh
echo env STAGE_HOST=shd22
```

Coding conventions for prolog programs:
The prolog program is invoked by the starter process. Once the starter process invokes the prolog program, the program obtains information about the job from environment variables.

Syntax:
prolog_program

Where *prolog_program* is the name of the prolog program as defined in the JOB_PROLOG keyword.

No arguments are passed to the program, but several environment variables are set. For more information on these environment variables, see "Run-time environment variables" on page 384.

The real and effective user ID of the prolog process is the LoadLeveler user ID. If the prolog program requires root authority, the administrator must write a secure C or Perl program to perform the desired actions. You should *not* use shell scripts with set uid permissions, since these scripts may make your system susceptible to security problems.

Return code values:
0 The job will begin.

If the prolog program is ended with a signal, the job does not begin and a message is written to the starter log.

Sample prolog programs:

- **Sample of a prolog program for korn shell:**

```
#!/bin/ksh
#
# Set up environment
set -a
. /etc/environment
. /.profile
export PATH="$PATH:/loctools/lladmin/bin"
export LOG="/tmp/$LOADL_STEP_OWNER.$LOADL_STEP_ID.prolog"
#
# Do set up based upon job step class
#
case $LOADL_STEP_CLASS in
    # A OSL job is about to run, make sure the osl filesystem is
    # mounted. If status is negative then filesystem cannot be
    # mounted and the job step should not run.
    "OSL")
      mount_osl_files >> $LOG
    if [ status = 0 ]
        then EXIT_CODE=1
      else
        EXIT_CODE=0
      fi
      ;;
# A simulation job is about to run, simulation data has to
# be made available to the job. The status from copy script must
# be zero or job step cannot run.
"sim")

      copy_sim_data >> $LOG
if [ status = 0 ]
        then EXIT_CODE=0
      else
        EXIT_CODE=1
      fi
      ;;
# All other job will require free space in /tmp, make sure
# enough space is available.
*)
```

```
      check_tmp >> $LOG
      EXIT_CODE=$?
      ;;
esac
# The job step will run only if EXIT_CODE == 0
exit $EXIT_CODE
```

- **Sample of a prolog program for C shell:**

```
#!/bin/csh
#
# Set up environment
source /u/loadl/.login
#
setenv PATH  "${PATH}:/loctools/lladmin/bin"
setenv LOG "/tmp/${LOADL_STEP_OWNER}.${LOADL_STEP_ID}.prolog"
#
# Do set up based upon job step class
#
switch ($LOADL_STEP_CLASS)
    # A OSL job is about to run, make sure the osl filesystem is
    # mounted. If status is negative then filesystem cannot be
    # mounted and the job step should not run.
    case "OSL":
      mount_osl_files >> $LOG
      if ($status < 0 ) then
        set EXIT_CODE = 1
      else
        set EXIT_CODE = 0
      endif
      breaksw
# A simulation job is about to run, simulation data has to
# be made available to the job. The status from copy script must
# be zero or job step cannot run.
case "sim":
    copy_sim_data >> $LOG
    if ($status == 0 ) then
      set EXIT_CODE = 0
    else
      set EXIT_CODE = 1
    endif
    breaksw
# All other job will require free space in /tmp, make sure
# enough space is available.
default:
    check_tmp >> $LOG
    set EXIT_CODE = $status
    breaksw
endsw

# The job step will run only if EXIT_CODE == 0
exit $EXIT_CODE
```

Coding conventions for epilog programs:
The installation-defined epilog program is invoked after a job step has completed. The purpose of the epilog program is to perform any required clean up such as unmounting file systems, removing files, and copying results. The exit status of both the prolog program and the job step is set in environment variables.

Syntax:

epilog_program

Where *epilog_program* is the name of the epilog program as defined in the **JOB_EPILOG** keyword.

No arguments are passed to the program but several environment variables are set. These environment variables are described in "Run-time environment variables" on page 384. In addition, the following environment variables are set for the epilog programs:

LOADL_PROLOG_EXIT_CODE
The exit code from the prolog program. This environment variable is set only if a prolog program is configured to run.

LOADL_USER_PROLOG_EXIT_CODE
The exit code from the user prolog program. This environment variable is set only if a user prolog program is configured to run.

LOADL_JOB_STEP_EXIT_CODE
The exit code from the job step.

Note: To interpret the exit status of the prolog program and the job step, convert the string to an integer and use the macros found in the **sys/wait.h** file. These macros include:
- WEXITSTATUS: gives you the exit code
- WTERMSIG: gives you the signal that terminated the program
- WIFEXITED: tells you if the program exited
- WIFSIGNALED: tells you if the program was terminated by a signal

The exit codes returned by the WEXITSTATUS macro are the valid codes. However, if you look at the raw numbers in **sys/wait.h**, the exit code may appear to be 256 times the expected return code. The numbers in **sys/wait.h** are the wait3 system calls.

Sample epilog programs:

- **Sample of an epilog program for korn shell:**

```
#!/bin/ksh
#
# Set up environment
set -a
. /etc/environment
. /.profile
export PATH="$PATH:/loctools/lladmin/bin"
export LOG="/tmp/$LOADL_STEP_OWNER.$LOADL_STEP_ID.epilog"
#
if [ [ -z $LOADL_PROLOG_EXIT_CODE ] ]
then
echo "Prolog did not run" >> $LOG
else
echo "Prolog exit code = $LOADL_PROLOG_EXIT_CODE" >> $LOG
fi
#
if [ [ -z $LOADL_USER_PROLOG_EXIT_CODE ] ]
  then
   echo "User environment prolog did not run" >> $LOG
  else
   echo "User environment exit code = $LOADL_USER_PROLOG_EXIT_CODE" >> $LOG
fi
#
if [ [ -z $LOADL_JOB_STEP_EXIT_CODE ] ]
  then
   echo "Job step did not run" >> $LOG
  else
   echo "Job step exit code = $LOADL_JOB_STEP_EXIT_CODE" >> $LOG
fi
#
#
# Do clean up based upon job step class
#
case $LOADL_STEP_CLASS in
```

```
  # A OSL job just ran, unmount the filesystem.
  "OSL")
    umount_osl_files >> $LOG
    ;;
  # A simulation job just ran, remove input files.
  # Copy results if simulation was successful (second argument
  # contains exit status from job step).
  "sim")
    rm_sim_data >> $LOG
    if [ $2 = 0 ]
      then copy_sim_results >> $LOG
    fi
    ;;
# Clean up /tmp
*)
  clean_tmp >> $LOG
  ;;
esac
```

- **Sample of an epilog program for C shell:**

```
#!/bin/csh
#
# Set up environment
source /u/loadl/.login
#
setenv PATH  "${PATH}:/loctools/lladmin/bin"
setenv LOG "/tmp/${LOADL_STEP_OWNER}.${LOADL_STEP_ID}.prolog"
#
if ( ${?LOADL_PROLOG_EXIT_CODE} ) then
echo "Prolog exit code = $LOADL_PROLOG_EXIT_CODE" >> $LOG
else
echo "Prolog did not run" >> $LOG
endif
#
if ( ${?LOADL_USER_PROLOG_EXIT_CODE} ) then
    echo "User environment exit code = $LOADL_USER_PROLOG_EXIT_CODE" >> $LOG
  else
    echo "User environment prolog did not run" >> $LOG
endif
#
if ( ${?LOADL_JOB_STEP_EXIT_CODE} ) then
    echo "Job step exit code = $LOADL_JOB_STEP_EXIT_CODE" >> $LOG
  else
    echo "Job step did not run" >> $LOG
endif
#
# Do clean up based upon job step class
#
switch ($LOADL_STEP_CLASS)
  # A OSL job just ran, unmount the filesystem.
  case "OSL":
    umount_osl_files >> $LOG
    breaksw
# A simulation job just ran, remove input files.
# Copy results if simulation was successful (second argument
# contains exit status from job step).
case "sim":
  rm_sim_data >> $LOG
  if ($argv{2} == 0 ) then
    copy_sim_results >> $LOG
  endif
  breaksw
# Clean up /tmp
default:
  clean_tmp >> $LOG
  breaksw
endsw
```

Using your own mail program

You can write a program to override the LoadLeveler default mail notification method. You can use this program, for example, to display your own messages to users when a job completes, or to automate tasks such as sending error messages to a network manager.

The syntax for the program is the same as it is for standard UNIX mail programs; the command is called with the following arguments:

- **-s** to indicate a subject.
- A pointer to a string containing the subject.
- A pointer to a string containing a list of mail recipients.

The mail message is taken from standard input.

To enable this program to replace the default mail notification method, use the **MAIL** keyword in the configuration file. For details about specific keyword syntax and use in the configuration, see Chapter 10, "Configuration keyword reference," on page 231.

Determining the frequency to use to run a job

You can write an installation exit that can determine the frequency to use to run a job. The first time a job is submitted, it runs in the nominal frequency. LoadLeveler gathers the hardware counters for the job and generates the energy tag. When the job runs again, LoadLeveler runs the specified installation exit program to determine the frequency to use to run the job. The energy tag name that was generated during the first run is passed to the user program by the **LL_ENERGY_TAG_NAME** environment variable. The user program can query the energy tag information by using the LoadLeveler API if needed.

The external frequency program is invoked by the Startd process before the job runs.

Syntax:

external_program

where:

external_program
: Is the name of the program as defined in the **EXT_ENERGY_POLICY_PROGRAM** keyword. No arguments are passed to the program.

The **LL_ENERGY_TAG_NAME** environment variable is exported to the program and can be used to query the energy data generated for this tag by LoadLeveler. For more information about the environment variable, see "Run-time environment variables" on page 384.

Return code values:

>0 The calculated frequency for running the job.

LoadLeveler uses the returned value as the frequency to use for running the job. If the program returns 0 or a negative value, the job will run at the nominal frequency.

Sample external frequency program:

The following is a sample C program, which returns the frequency to use to run a job:

```
/*
 * ------------------------------------------------------------------------
 * Determine the frequency of the job by the user program
 * ------------------------------------------------------------------------
 */

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <string.h>
#include <assert.h>
#include <errno.h>
#include <stdint.h> // for uint64_t
#include "llrapi.h"

int
main(int argc, char *argv[]) {
    int frequency = -1;

    // query the energy tag info if needed.
        llr_resmgr_handle_t *rm_handle;
        llr_query_handle_t *q_handle = NULL;
        llr_element_t *err_object;
        llr_element_t **object_list;
        llr_element_t *etag = NULL;
        llr_data_list_t policy_list;
        llr_etag_policy policy;
        void* p = (void *)&policy;
        int rc;
        int i;
        llr_query_filter_t etag_q_filter[1];
        int etag_filter_cnt = 1;
        char **etag_name = (char**)malloc(2*sizeof(char*));
        bzero(etag_name, 2* sizeof(char *));
        assert(argc > 1);
        *etag_name = strdup(argv[1]);
        etag_q_filter[0].filter_type = LLR_QUERY_ETAGNAME; /* take tag name as the query filter */
        etag_q_filter[0].filter_data = etag_name;

        err_object = NULL;
        rc = llr_init_resmgr(LLR_API_VERSION, &rm_handle, &err_object);
        if (rc != LLR_API_OK) {
                fprintf(stderr, "Resource manager API can not be intialized.\n");
                if (err_object) {
                        llr_error(&err_object, LLR_ERROR_PRINT_STDERR);
                }
                exit(-1);
        }

        rc = llr_query_set(rm_handle, &q_handle, LLR_ENERGYTAG_QUERY, etag_filter_cnt, \
        etag_q_filter, &err_object);
        if (rc != LLR_API_OK) {
                fprintf(stderr, "Failed to set up job query.\n");
                if (err_object) {
                        llr_error(&err_object, LLR_ERROR_PRINT_STDERR);
                }
                exit(-1);
        }

        rc =  llr_query_get_data(rm_handle, q_handle, LLR_QUERY_RESOURCE_MANAGER, NULL, \
        &object_list, &err_object);
        if (rc != LLR_API_OK) {
                if (err_object) {
                        llr_error(&err_object, LLR_ERROR_PRINT_STDERR);
                }
                fprintf(stderr, "llr_query_get_data failed.\n");
                exit(-1);
        }

        if (object_list==NULL || object_list[0]==NULL) {
                fprintf(stdout, "No policy record found.\n");
                exit(0);
        }

        memset(&policy, 0, sizeof(policy));
        for (i=0; object_list&&(etag = object_list[i])!=NULL; i++) {
            rc += llr_get_data(rm_handle, etag, LLR_ETagGetPolicyList, &policy_list, \
```

```
            &err_object); p = &policy;
            rc += llr_get_data(rm_handle, &policy_list, LLR_ETagGetFirstPolicy, (void*)&p, \
            &err_object); while (p != NULL) {
                 fprintf(stdout, "user=%s, etag_name=%s, freq=%d KHZ, est. energy=%f KWH, \
                 est. time=%d s, perfd=%f%%, energy saving=%f%%\n", policy.username, \
                 policy.energy_tag_name, policy.frequency, policy.predict_power, \
                 policy.predict_elapse_time, policy.perf_degrad_pct*100, \
                 policy.energy_saving_pct*100); memset(&policy, 0, sizeof(policy))\
                 ;p = &policy; rc += llr_get_data(rm_handle, &policy_list, \
                 LLR_ETagGetNextPolicy,(void*)&p, &err_object);
            }

            err_object = NULL;
            if (rc != LLR_API_OK) {
                 if (err_object) {
                        llr_error(&err_object, LLR_ERROR_PRINT_STDERR);
                 }
                 fprintf(stderr, "llr_get_data failed.\n");
                 exit(-1);
            }
        }
        rc =  llr_query_free_data(rm_handle, &q_handle, &err_object);
        llr_free_resmgr(&rm_handle, &err_object);

// calculate the frequency by user logic
frequency = valid_frequency;

 return frequency;
}
```

To compile the source code, use this command:

```
gcc -o myengprog -lllrapi -I /opt/ibmll/LoadL/resmgr/full/include myengprog.c
```

Chapter 5. Defining LoadLeveler resources to administer

After installing LoadLeveler, you can customize it by modifying the **administration** file, or if the database configuration option is used, by using the **llconfig** or **llrconfig** command with the **-s** option, or by using the configuration editor to modify the tables for the machines, machine_groups, classes, users, and groups.

For file-based configuration, the administration file optionally lists and defines the machines in the LoadLeveler cluster and the characteristics of classes, users, and groups.

LoadLeveler does not prevent you from having multiple copies of administration files, but you need to be sure to update all the copies whenever you make a change to one. Having only one administration file prevents any confusion.

Table 16 lists the LoadLeveler resources you may define by modifying the administration file.

Table 16. Roadmap of tasks for modifying the LoadLeveler administration file

To learn about:	Read the following:
Defining LoadLeveler resources to administer	• "Defining machines" • "Dynamic adapter discovery" on page 93 • "LoadLeveler adapter and node status monitoring" on page 94 • "Defining classes" on page 94 • "Defining users" on page 102 • "Defining groups" on page 103 • "Defining clusters" on page 104
Correctly specifying administration file keywords	Chapter 11, "Administration keyword reference," on page 293

Defining machines

Characteristics of a machine may be defined in a machine stanza, a machine group stanza, or a machine sub-stanza of a machine group.

A machine_group stanza groups together machines with similar characteristics. Using machine_groups reduces the need to specify each machine in the cluster in the administration file because each machine_group stanza can specify a **machine_list** that supports machine range expressions similar to the syntax supported by xCAT.

If you do not specify a machine or machine_group stanza for a machine in the cluster, the machines can still communicate with one another and jobs are scheduled on the machine but the machine is assigned the default values specified in the default machine_group stanza. If there is no default stanza, the machine is assigned default values set by LoadLeveler.

If you set the **MACHINE_AUTHENTICATE** keyword to true in the configuration file, then for each machine that LoadLeveler includes in the cluster you must either

create a machine stanza or include the machine in a machine_group using either a machine substanza or the **machine_list** keyword.

In the LoadLeveler configuration, machine names are stored in lower case, so all references to the machine names must be in lower case.

Planning considerations for defining machines

There are several matters to consider before customizing the administration file. Before customizing the administration file, consider the following:

- **Node availability**

  Some workstation owners might agree to accept LoadLeveler jobs only when they are not using the workstation themselves. Using LoadLeveler keywords, these workstations can be configured to be available at designated times only.

- **Common name space**

  To run jobs on any machine in the LoadLeveler cluster, a user needs the same uid (the user ID number for a user) and gid (the group ID number for a group) on every machine in the cluster.

  For example, if there are two machines in your LoadLeveler cluster, *machine_1* and *machine_2*, user `john` must have the same user ID and login group ID in the **/etc/passwd** file on both machines. If user `john` has user ID 1234 and login group ID 100 on *machine_1*, then user `john` must have the same user ID and login group ID in **/etc/passwd** on *machine_2*. (LoadLeveler requires a job to run with the same group ID and user ID of the person who submitted the job.)

  If you do not have a user ID on one machine, your jobs will not run on that machine. Also, many commands, such as **llq**, will not work correctly if a user does not have a user ID on the central manager machine.

  However, there are cases where you may choose to not give a user a login ID on a particular machine. For example, a user does not need an ID on every submit-only machine; the user only needs to be able to submit jobs from at least one such machine. Also, you may choose to restrict a user's access to a Schedd machine that is not a public scheduler; again, the user only needs access to at least one Schedd machine.

- **Resource handling**

  Some nodes in the LoadLeveler cluster might have special software installed that users might need to run their jobs successfully. You should configure LoadLeveler to distinguish those nodes from other nodes using, for example, machine features.

Machine_group stanza format and keyword summary

Machine_group stanzas take the following format. The default values for keywords appear in bold:

```
label: {
   type = machine_group
   adapter_list = adapter_name...
   class = class_name(count) class_name(count) ... class_name(count)
   cpu_speed_scale = true | false
   feature = feature_name...
   island = name
   machine_list = range_expression
   machine_mode = batch | interactive | general
   master_node_exclusive = true | false
   max_jobs_scheduled = number
   max_starters = number
   dstg_max_starters = number
```

```
   name_server = list
   pool_list = pool_numbers
   power_management_policy =start_time;duration | off
   prestarted_starters = number
   region = region_name
   reservation_permitted = true | false
   resources = name(count) name(count) ... name(count)
   schedd_fenced = true | false
   schedd_host = true | false
   schedd_runs_here = true | false
   speed = number
   startd_runs_here = true | false
   submit_only = true | false
}
```

Notes:

1. The central manager keyword is not supported in the machine_group stanza. Instead, the **central_manager_list** keyword should be specified in the configuration file or database.
2. Each of these machine stanza keywords apply to all machines within the group. For example, specifying **reservation_permitted = false** in a machine_group stanza means that every machine in that machine group has **reservation_permitted** set to false.
3. The **machine_list** keyword is optional when there are machine substanzas, otherwise **machine_list** must be present.
4. A machine can belong to only one machine_group. Machines can also be defined in separate machine stanzas outside of any machine_group.

Machine substanza format and keyword summary

The following example shows a substanza of type machine. The default values for keywords appear in bold:

```
label: {
  type = machine_group
  ...
  label: {
  type = machine
  feature = feature_name...
  master_node_exclusive = true | false
  max_jobs_scheduled = number
  schedd_fenced = true | false
  schedd_host = true | false
  schedd_runs_here = true | false
  startd_runs_here = true | false
 }
}
```

Notes:

1. Not all keywords that are permitted in machine and machine_group stanzas may be specified in a machine substanza. If it is necessary to override a machine_group setting for a machine, then that machine should be defined in its own machine stanza outside of any machine_group.
2. A machine may appear as a machine substanza and in the machine_list within the same machine_group stanza.

Machine stanza format and keyword summary

Machine stanzas take the following format. The default values for keywords appear in bold:

```
label: type = machine
  adapter_list = adapter_name...
  class = class_name(count) class_name(count) ... class_name(count)
  cpu_speed_scale = true | false
  feature = feature_name...
  island = name
  machine_mode = batch | interactive | general
  master_node_exclusive = true | false
  max_jobs_scheduled = number
  max_starters = number
  dstg_max_starters = number
  name_server = list
  pool_list = pool_numbers
  power_management_policy =start_time;duration | off
  prestarted_starters = number
  region = region_name
  reservation_permitted = true | false
  resources = name(count) name(count) ... name(count)
  schedd_fenced = true | false
  schedd_host = true | false
  schedd_runs_here = true | false
  speed = number
  startd_runs_here = true | false
  submit_only = true | false
```

Default values for machine_group and machine stanzas

A special machine_group stanza with the name of default can be specified to define the values for keywords for all other machine_group and machine stanzas. Any keyword not explicitly defined in the default machine_group stanza is assigned a default value by LoadLeveler. The rules governing these default values include:

- A machine inherits the attributes from its machine_group. If a machine does not belong to a machine_group, it inherits the attributes from the default machine_group.
- A default machine_group stanza must come before any other machine_group stanzas.
- The **machine_list** keyword and machine substanzas are not allowed in a default machine_group stanza.
- If there are any machine_group stanzas present in the administration file at all, then a default machine stanza cannot be specified (it will be considered an error). Both machine and machine_group stanzas inherit the values specified in the default machine_group stanza.
- A nondefault machine_group stanza serves as the default stanza of all the machines its **machine_list** covers.
- The explicitly defined machine substanza will also use the machine_group stanza's keywords as default, even if it is not included in the range the **machine_list** represents.

Examples of machine_group and machine stanzas

These machine stanza examples may apply to your situation.

- **Example 1**

  This example sets up a submit-only node. Note that the **submit-only** keyword in the example is set to **true**, while the **schedd_host** keyword is set to **false**. You must also ensure that you set the **schedd_host** to **true** on at least one other node in the cluster.

```
#
machine_b: type = machine
 schedd_host = false       # not a scheduling machine
 submit_only = true        # submit only machine
```

- **Example 2**

```
default: {
 type = machine_group
 machine_mode = general
 pool_list = 1 7
 startd_runs_here = true
 schedd_runs_here = false
}

MG1: {
 type = machine_group
 schedd_host = true
 resources = ConsumableCpus(all)
 machine_list = x330n01-x330n99,-x330n10,-x330n20
 x330n50: {
   type = machine
   schedd_runs_here = true
   startd_runs_here = false
  }
}

MG2: {
 Type = machine_group
 Pool_list = 1
 resources = ConsumableCpus(4)
 Machine_list = x330n10,x330n20
}
```

In this example, x330n50 is the only machine in the cluster where a Schedd daemon will be started, and no Startd daemon will run there. X330n10 and x330n20 are defined as belonging only to pool 1 and having only 4 ConsumableCpus available, while all other machines in the cluster belong to both pool 1 and pool 7 and all machines in MG1 have all of their CPUs available as ConsumableCpus.

Dynamic adapter discovery

Adapters are dynamically discovered by the LoadLeveler startd daemon by querying the system configuration and the Protocol Network Services Daemon (PNSD). Startd sends the adapter configuration to the negotiator, resource manager, and region manager daemons on start up and reconfiguration. User space jobs will be supported using InfiniBand or the Host Fabric Interface (HFI) adapters, if PNSD is installed on the nodes. If PNSD is not installed, all adapters will be treated as Ethernet adapters.

The Startd daemon polls the system configuration and PNSD every (POLLS_PER_UPDATE * POLLING_FREQUENCY) seconds to pick up any new information. Any changes discovered are sent to the negotiator, resource manager, and region manager.

If only certain adapter interfaces are to be used for a machine, then the **adapter_list** keyword under the machine or **machine_group** stanza can be used to list the adapter interface names in the order that will be used for scheduling jobs. If this keyword is not specified in the machine or machine group configuration, then all discovered adapters will be used.

Notes for InfiniBand adapters:

1. LoadLeveler distributes the switch adapter windows of the InfiniBand adapter equally among its ports and the allocation is not adjusted should all of the resources on one port be consumed.
2. If one InfiniBand port is in use exclusively, no other ports on the InfiniBand adapter can be used for any other job.
3. Because InfiniBand adapters do not support rCxt blocks, jobs that request InfiniBand adapters and rCxt blocks with the **rcxtblocks** keyword on the network statement will remain in the idle state. You can use the **llstatus -a** command to see rCxt blocks on adapters (see the **llstatus** command in *LoadLeveler: Command and API Reference* for more information).

LoadLeveler adapter and node status monitoring

Machine and adapter configuration and status changes will be detected by the region manager and the startd daemons. If the region manager daemon is not configured, the adapter and node status will only come from the configuration information available to the startd daemon and will not reflect the actual connectivity of the adapter or node.

Note: The region manager node must have similar connectivity to the network as the executing machine it manages, so that all of its configured network interfaces are able to connect to all of the executing machine's network interfaces.

The startd daemon generates its adapter configuration and local status and sends the information to the region manager, central manager, and resource manager. Adapter evaluations are done by the startd during the **polls_per_update * polling_frequency** intervals. The polling keywords will trigger how often the adapter information is updated. Node and adapter information and status will be sent to the daemons if changes were detected by the startd. The startd also sends heartbeats to the region manager over each of its configured network interfaces if a region is configured.

The region manager maintains the machine and adapter status for all the nodes in its managed region. The region manager will mark the adapter down after a period of **adapter_heartbeat_interval * adapter_heartbeat_retries** if no heartbeat is received. When an adapter comes up, the region manager will receive the heartbeat from the startd and will immediately mark it as up. The region manager will send heartbeat status changes to the central manager and the resource manager.

Defining classes

The information in a class stanza defines characteristics for that class.

These characteristics can include the quantities of consumable resources that may be used by a class per machine or cluster.

Within a class stanza, you can have optional user substanzas that define policies that apply to a user's job steps that need to use this class. For more information about user substanzas, see “Defining user substanzas in class stanzas” on page 99. For information about user stanzas, see “Defining users” on page 102.

Using limit keywords

A limit is the amount of a resource that a job step or a process is allowed to use. (A process is a dispatchable unit of work.) A job step may be made up of several processes.

Limits include both a **hard limit** and a **soft limit**. When a hard limit is exceeded, the job is usually terminated. When a soft limit is exceeded, the job is usually given a chance to perform some recovery actions. Limits are enforced either per process or per job step, depending on the type of limit. For parallel jobs steps, which consist of multiple tasks running on multiple machines, limits are enforced on a per task basis.

The class stanza includes the **limit** keywords shown in Table 17, which allow you to control the amount of resources used by a job step or a job process.

Table 17. Types of limit keywords

Limit	How the limit is enforced
as_limit	Per process
ckpt_time_limit	Per job step
core_limit	Per process
cpu_limit	Per process
data_limit	Per process
default_wall_clock_limit	Per job step
file_limit	Per process
job_cpu_limit	Per job step
locks_limit	Per process
memlock_limit	Per process
nofile_limit	Per process
nproc_limit	Per user
rss_limit	Per process
stack_limit	Per process
wall_clock_limit	Per job step

For example, a common limit is the **cpu_limit**, which limits the amount of CPU time a single process can use. If you set **cpu_limit** to five hours and you have a job step that forks five processes, each process can use up to five hours of CPU time, for a total of 25 CPU hours. Another limit that controls the amount of CPU used is **job_cpu_limit**. For a serial job step, if you impose a **job_cpu_limit** of five hours, the entire job step (made up of all five processes) cannot consume more than five CPU hours. For information on using this keyword with parallel jobs, see "Job command file keyword descriptions" on page 335.

You can specify limits in either the class stanza of the administration file or in the job command file. The lower of these two limits will be used to run the job even if the system limit for the user is lower. For more information, see:

- "Enforcing limits"
- "Administration keyword descriptions" on page 298 or "Job command file keyword descriptions" on page 335

Enforcing limits

LoadLeveler depends on the underlying operating system to enforce process limits. Users should verify that a process limit such as **rss_limit** is enforced by the operating system, otherwise setting it in LoadLeveler will have no effect.

Exceeding job step limits: When a hard limit is exceeded LoadLeveler sends a *non-trappable* signal (except in the case of a parallel job) to the process group that LoadLeveler created for the job step. When a soft limit is exceeded, LoadLeveler sends a *trappable* signal to the process group. Any job application that intends to trap a signal sent by LoadLeveler must ensure that all processes in the process group set up the appropriate signal handler.

All processes in the job step normally receive the signal. The exception to this rule is when a child process creates its own process group. That action isolates the child's process, and its children, from any signals that LoadLeveler sends. Any child process creating its own process group is still known to process tracking. So, if process tracking is enabled, all the child processes are terminated when the main process terminates.

Table 18 summarizes the actions that the **LoadL_starter** daemon takes when a job step limit is exceeded.

Table 18. Enforcing job step limits

Type of Job	When a Soft Limit is Exceeded	When a Hard Limit is Exceeded
Serial	SIGXCPU or SIGKILL issued	SIGKILL issued
Parallel	SIGXCPU issued to both the user program and to the parallel daemon	SIGTERM issued

On systems that do not support SIGXCPU, LoadLeveler does not distinguish between hard and soft limits. When a soft limit is reached on these platforms, LoadLeveler issues a SIGKILL.

Enforcing per process limits: For per process limits, what happens when your job reaches and exceeds either the soft limit or the hard limit depends on the operating system you are using. When a job forks a process that exceeds a per process limit, such as the CPU limit, the operating system (not LoadLeveler) terminates the process by issuing a SIGXCPU. As a result, you will not see an entry in the LoadLeveler logs indicating that the process exceeded the limit. The job will complete with a 0 return code. LoadLeveler can only report the status of any processes it has started.

If you need more specific information, refer to your operating system documentation.

How LoadLeveler uses hard limits: Consider these details on how LoadLeveler uses hard limits. See Table 19 for more information on specifying limits.

Table 19. Setting limits

If the hard limit is:	Then LoadLeveler does the following:
Set in both the class stanza and the job command file	Smaller of the two limits is taken into consideration. If the smaller limit is the job limit, the job limit is then compared with the user limit set on the machine that runs the job. The smaller of these two values is used. If the limit used is the class limit, the class limit is used without being compared to the machine limit.
Not set in either the class stanza or the job command file	User per process limit set on the machine that runs the job is used.

Table 19. Setting limits (continued)

If the hard limit is:	Then LoadLeveler does the following:
Set in the job command file and is less than its respective job soft limit	The job is not submitted.
Set in the class stanza and is less than its respective class stanza soft limit	Soft limit is adjusted downward to equal the hard limit.
Specified in the job command file	Hard limit must be greater than or equal to the specified soft limit and less than or equal to the limit set by the administrator in the class stanza of the administration file. Note: If the per process limit is not defined in the administration file and the hard limit defined by the user in the job command file is greater than the limit on the executing machine, then the hard limit is set to the machine limit.

Allowing users to use a class

In a class stanza, you may define a list of users or a list of groups to identify those who may use the class. To do so, use the **include_users** or **include_groups** keyword, respectively, or you may use both keywords. If you specify both keywords, a particular user must satisfy both the **include_users** and the **include_groups** restrictions for the class. This requirement means that a particular user must be defined not only in a User stanza in the administration file, but also in one of the following ways:

- The user's name must appear in the **include_users** keyword in a Group stanza whose name corresponds to a name in the **include_groups** keyword of the Class stanza.
- The user's name must appear in the **include_groups** keyword of the Class stanza. For information about specifying a user name in a group list, see the **include_groups** keyword description in "Administration keyword descriptions" on page 298.

Class stanza format and keyword summary

Class stanzas are optional. Class stanzas take the following format. Default values for keywords appear in bold.

```
label: type = class
admin= list
as_limit= hardlimit,softlimit
ckpt_time_limit = hardlimit,softlimit
class_comment = "string"
collective_groups = number
core_limit = hardlimit,softlimit
cpu_limit = hardlimit,softlimit
data_limit = hardlimit,softlimit
default_network.protocol = type[, usage[, mode[,comm_level[, instances=<number \
 |max> [, rcxtblocks=number]]]]]
default_resources = name(count) name(count)...name(count)
default_node_resources = name(count) name(count)...name(count)
default_wall_clock_limit = hardlimit,softlimit
endpoints = number
env_copy = all | master
exclude_bg = list
exclude_groups = list
exclude_users = list
```

```
file_limit = hardlimit,softlimit
imm_send_buffers = number
include_bg = list
include_groups = list
include_users = list
job_cpu_limit = hardlimit,softlimit
locks_limit = hardlimit,softlimit
master_node_requirement = true | false
max_node = number
max_protocol_instances = number
max_resources = name(count) name(count)...name(count)
max_node_resources = name(count) name(count)...name(count)
max_top_dogs = number
max_total_tasks = number
maxjobs = number
memlock_limit = hardlimit,softlimit
nice = value
nofile_limit = hardlimit,softlimit
nproc_limit = hardlimit,softlimit
priority = number
rss_limit = hardlimit,softlimit
restart = yes | no
smt =  | as_is
stack_limit = hardlimit,softlimit
striping_with_minimum_networks = true | false
total_tasks = number
wall_clock_limit = hardlimit,softlimit
```

Examples: Class stanzas

Any of these class stanza examples may apply to your situation.

- **Example 1: Creating a class that excludes certain users**

```
class_a: type=class                # class that excludes users
priority=10                        # ClassSysprio
exclude_users=green judy           # Excluded users
```

- **Example 2: Creating a class for small-size jobs**

```
small:  type=class                                 # class for small jobs
priority=80                                        # ClassSysprio (max=100)
cpu_limit=00:02:00                                 # 2 minute limit
data_limit=30mb                                    # max 30 MB data segment
default_resources=ConsumbableVirtualMemory(10mb) # resources consumed by each
ConsumableCpus(1) resA(3) floatinglicenseX(1)      # task of a small job step if
                                                   # resources are not explicitly
                                                   # specified in the job command file
ckpt_time_limit=3:00,2:00                          # 3 minute hardlimit,
                                                   # 2 minute softlimit
core_limit=10mb                                    # max 10 MB core file
file_limit=50mb                                    # max file size 50 MB
stack_limit=10mb                                   # max stack size 10 MB
rss_limit=35mb                                     # max resident set size 35 MB
include_users = bob sally                          # authorized users
```

- **Example 3: Creating a class for medium-size jobs**

```
medium: type=class             # class for medium jobs
priority=70                    # ClassSysprio
cpu_limit=00:10:00             # 10 minute run time limit
data_limit=80mb,60mb           # max 80 MB data segment
                               # soft limit 60 MB data segment
ckpt_time_limit=5:00,4:30      # 5 minute hardlimit,
                               # 4 minute 30 second softlimit to checkpoint
core_limit=30mb                # max 30 MB core file
file_limit=80mb                # max file size 80 MB
stack_limit=30mb               # max stack size 30 MB
rss_limit=100mb                # max resident set size 100 MB
job_cpu_limit=1800,1200        # hard limit is 30 minutes,
                               # soft limit is 20 minutes
```

- **Example 4: Creating a class for large-size jobs**

```
large:  type=class             # class for large jobs
priority=60                    # ClassSysprio
cpu_limit=00:10:00             # 10 minute run time limit
data_limit=120mb               # max 120 MB data segment
default_resources=ConsumableVirtualMemory(40mb)         # resources consumed
ConsumableCpus(2) resA(8) floatinglicenseX(1) resB(1)   # by each task of
                               # a large job step if resources are not
                               # explicitly specified in the job command file
ckpt_time_limit=7:00,5:00      # 7 minute hardlimit,
                               # 5 minute softlimit to checkpoint
core_limit=30mb                # max 30 MB core file
file_limit=120mb               # max file size 120 MB
stack_limit=unlimited          # unlimited stack size
rss_limit=150mb                # max resident set size 150 MB
job_cpu_limit = 3600,2700      # hard limit 60 minutes
                               # soft limit 45 minutes
wall_clock_limit=12:00:00,11:59:55 # hard limit is 12 hours
```

- **Example 5: Creating a class for master node machines**

```
sp-6hr-sp:  type=class         # class for master node machines
priority=50                    # ClassSysprio (max=100)
ckpt_time_limit=25:00,20:00    # 25 minute hardlimit,
                               # 20 minute softlimit to checkpoint
cpu_limit = 06:00:00           # 6 hour limit
job_cpu_limit = 06:00:00       # hard limit is 6 hours
core_limit = 1mb               # max 1MB core file
master_node_requirement = true # master node definition
```

Defining user substanzas in class stanzas

In a class stanza, you might define user substanzas using the same syntax as you would for any stanza in the LoadLeveler administration file.

A user substanza within a class stanza defines policies that apply to job steps submitted by that user and belonging to that class. User substanzas are optional and are independent of user stanzas (for information about user stanzas, see "Defining users" on page 102).

Class stanzas that contain user substanzas have the following format:

```
label: {
       type = class
       label: {
              type = user
              maxidle = number
              maxjobs = number
              maxqueued = number
              max_total_tasks = number
        }
}
```

When defining substanzas within other stanzas, you must use opening and closing braces ({ and }) to mark the beginning and the end of the stanza and substanza. The only keywords that are supported in a user substanza are **type** (required), **maxidle**, **maxjobs**, **maxqueued**, and **max_total_tasks**. For detailed descriptions of these keywords, see "Administration keyword descriptions" on page 298.

Examples: Substanzas

Any of these substanza examples may apply to your situation.

In the following example, the default machine and class stanzas do not require braces, but the parallel class stanza does require them. Without braces to open and close the parallel stanza, it would not be clear that the default user and **dept_head** user stanza belong to the parallel class:

```
default:
      type = machine
      schedd_host = true

default:
      type = class
      wall_clock_limit = 60:00,30:00

parallel: {
      type = class

      # Allow at most 50 running jobs for class parallel
      maxjobs = 50

      # Allow at most 10 running jobs for any single
      # user of class parallel
      default: {
            type = user
            maxjobs = 10

      }

      # Allow user dept_head to run as many as 20 jobs
      # of class parallel
      dept_head: {type = user
            maxjobs = 20

      }
}

dept_head: type = user
      maxjobs = 30
```

When user substanzas are used in class stanzas, a default user substanza can be defined. Each class stanza can have its own default user substanza, and even the default class stanza can have a default user substanza. In this example, the default user substanza in the default class indicates that for any combination of class and user, the limits **maxidle=20** and **maxqueued=30** apply, and that **maxjobs** and **max_total_tasks** are unlimited. Some of these values are overridden in the physics class stanza. Here is an example of how class stanzas can be configured:

```
default: {
      type = class
      default: {
            type = user
            maxidle = 20
            maxqueued = 30
            maxjobs = -1
            max_total_tasks = -1
      }
}
physics: {
      type = class
      default: {
            type = user
            maxjobs = 10
            max_total_tasks = 128
      }
      john: {
            type = user
            maxidle = 10
```

```
            maxjobs = 14
      }
      jane: {
            type = user
            max_total_tasks = 192
      }
}
```

In the following example, the physics stanza shows which values are inherited from which stanzas:

```
physics: {
         type = class
         default: {
               type = user
               # inherited from default class, default user
               # maxidle = 20

               # inherited from default class, default user
               # maxqueued = 30

               # overrides value of -1 in default class, default user
               maxjobs = 10

               # overrides value of -1 in default class, default user
               max_total_tasks = 128
         }
         john: {
               type = user
               # overrides value of 10 in default user
               maxidle = 10

               # inherited from default user, which was inherited
               # from default class, default user
               # maxqueued = 30

               # overrides value of 10 in default user
               maxjobs = 14

               # inherited from default user
               # max_total_tasks = 128
        }

        jane: {
              type = user
              # inherited from default user, which was inherited
              # from default class, default user
              # maxidle = 20

              # inherited from default user, which was inherited
              # from default class, default user
              # maxqueued = 30

              # inherited from default user
              # maxjobs = 10

              # overrides value of 128 in default user
              max_total_tasks = 192
        }
}
```

Any user other than john and jane who submits jobs of class physics is subject to the constraints in the default user substanza in the physics class stanza. Should john or jane submit jobs of any class other than physics, they are subject to the constraints in the default user substanza in the default class stanza.

In addition to specifying a default user substanza within the default class stanza, an administrator can specify other user substanzas in the default class stanza. It is important to note that all class stanzas will inherit all user substanzas from the default class stanza.

Note: An important rule to understand is that a user substanza within a class stanza will inherit its values from the user substanza in the default class stanza first, if a substanza for that user is present. The next location a user substanza inherits values from is the default user substanza within the same class stanza.

When no default stanzas or substanzas are provided, the LoadLeveler default for all four keywords is -1 or unlimited.

If a user substanza is provided for a user on the class **exclude_users** list, **exclude_users** takes precedence and the user substanza will be effectively ignored because that user cannot use the class at all. On the other hand, when **include_users** is used in a class, the presence of a user substanza implies that the user is permitted to use the class (it is as if the user were present on the **include_users** list).

Defining users

The information specified in a user stanza defines the characteristics of that user. You can have one user stanza for each user but this is not necessary. If an individual user does not have their own user stanza, that user uses the defaults defined in the default user stanza.

User stanza format and keyword summary

User stanzas take the following format:

```
label: type = user
account = list
default_class = list
default_group = group name
default_interactive_class = class name
env_copy = all | master
fair_shares = number
max_node = number
max_reservation_duration = number
max_reservation_expiration = number
max_reservations = number
max_total_tasks = number
maxidle = number
maxjobs = number
maxqueued = number
priority = number
reservation_type = all | flex | none
total_tasks = number
```

For more information about the keywords listed in the user stanza format, see Chapter 11, "Administration keyword reference," on page 293.

Examples: User stanzas

Any of the following user stanzas may apply to your situation.

- **Example 1**

  In this example, user `fred` is being provided with a user stanza. User `fred`'s jobs will have a user priority of 100. If user `fred` does not specify a job class in the

job command file, the default job class **class_a** will be used. In addition, he can have a maximum of 15 jobs running at the same time.

```
# Define user stanzas
fred:  type = user
priority = 100
default_class = class_a
maxjobs = 15
```

- **Example 2**

  This example explains how a default interactive class for a parallel job is set by presenting a series of user stanzas and class stanzas. This example assumes that users do not specify the LOADL_INTERACTIVE_CLASS environment variable.

```
default: type =user
         default_interactive_class = red
         default_class = blue

carol:   type = user
         default_class = single double
         default_interactive_class = ijobs

steve:   type = user
         default_class = single double

ijobs:   type = class
         wall_clock_limit = 08:00:00

red:     type = class
         wall_clock_limit = 30:00
```

  If the user Carol submits an interactive job, the job is assigned to the default interactive class called **ijobs**. The job is assigned a wall clock limit of 8 hours. If the user Steve submits an interactive job, the job is assigned to the **red** class from the default user stanza. The job is assigned a wall clock limit of 30 minutes.

- **Example 3**

  In this example, Jane's jobs have a user priority of 50, and if she does not specify a job class in her job command file the default job class **small_jobs** is used. This user stanza does not specify the maximum number of jobs that Jane can run at the same time so this value defaults to the value defined in the default stanza. Also, suppose Jane is a member of the primary UNIX group “staff.” Jobs submitted by Jane will use the default LoadLeveler group “staff.” Lastly, Jane can use three different account numbers.

```
# Define user stanzas
jane:  type = user
priority = 50
default_class = small_jobs
default_group = Unix_Group
account = dept10 user3 user4
```

Defining groups

LoadLeveler groups are another way of granting control to the system administrator.

Although a LoadLeveler group is independent from a UNIX group, you can configure a LoadLeveler group to have the same users as a UNIX group by using the **include_users** keyword. If you do not specify a value for the group keyword in the job command file, the default group for the user is used. If a default group is not defined for the user, LoadLeveler uses the group, **No_Group**.

Group stanza format and keyword summary

The information specified in a group stanza defines the characteristics of that group. Group stanzas are optional and take the following format:

```
label: type = group
admin = list
env_copy = all | master
fair_shares = number
exclude_users = list
include_users = list
max_node = number
max_reservation_duration = number
max_reservation_expiration = number
max_reservations = number
max_total_tasks = number
maxidle = number
maxjobs = number
maxqueued = number
priority = number
reservation_type = all | flex | none
total_tasks = number
```

For more information about the keywords listed in the group stanza format, see Chapter 11, "Administration keyword reference," on page 293.

Examples: Group stanzas

Any of the following group stanzas may apply to your situation.

- **Example 1**

  In this example, the group name is **department_a**. The jobs issued by users belonging to this group will have a priority of 80. There are three members in this group.

  ```
  # Define group stanzas
  department_a:  type = group
  priority = 80
  include_users = susann holly fran
  ```

- **Example 2**

  In this example, the group called **great_lakes** has five members and these user's jobs have a priority of 100:

  ```
  # Define group stanzas
  great_lakes:  type = group
  priority = 100
  include_users = huron ontario michigan erie superior
  ```

Defining clusters

The cluster stanza defines the LoadLeveler multicluster environment.

Any cluster that wants to participate in the multicluster must have cluster stanzas defined for all clusters with which the local cluster interacts. If you have a cluster stanza defined, LoadLeveler is configured to be in the multicluster environment.

Cluster stanza format and keyword summary

Cluster stanzas are optional. Cluster stanzas take the following format. Default values for keywords appear in bold.

The cluster stanza label must define a unique cluster name within the multicluster environment.

```
label: type = cluster
exclude_classes = class_name[(cluster_name)] ...
exclude_groups = group_name[(cluster_name)] ...
exclude_users = user_name[(cluster_name)] ...
inbound_hosts = hostname[(cluster_name)] ...
inbound_schedd_port = port_number
include_classes = class_name[(cluster_name)] ...
include_groups = group_name[(cluster_name)] ...
include_users = user_name[(clustername)] ...
local = true | false
multicluster_security = SSL
outbound_hosts = hostname[(cluster_name)] ...
secure_schedd_port = port_number
ssl_cipher_list = cipher_list
```

Examples: Cluster stanzas

Any of the following cluster stanzas may apply to your situation.

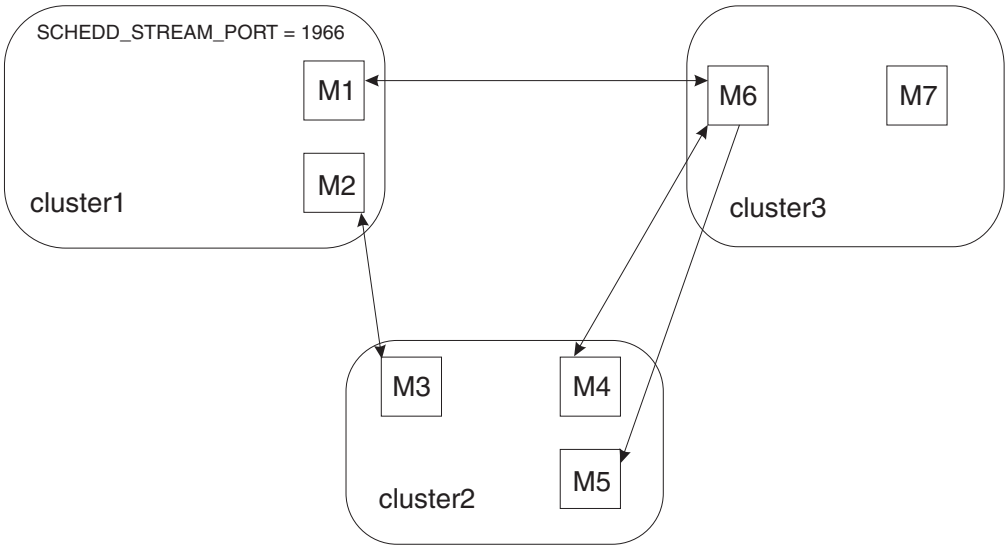

Figure 11. Multicluster Example

Figure 11 shows a simple multicluster with three clusters defined as members. Cluster1 has defined an alternate port number for the Schedds running in its cluster by setting the **SCHEDD_STREAM_PORT = 1966**. All of the other clusters need to define what port to use when connecting to the inbound Schedds of cluster1 by specifying the **inbound_schedd_port = 1966** keyword in the cluster1 stanza. Cluster2 has a single machine connected to cluster1 and 2 machines connected to cluster3. Cluster3 has a single machine connected to both cluster2 and cluster1. Each cluster would set the **local** keyword to **true** for their cluster stanza in the cluster's administration file.

Multicluster with 3 clusters defined as members

```
cluster1: type=cluster
          outbound_hosts = M2(cluster2) M1(cluster3)
          inbound_hosts = M2(cluster2) M1(cluster3)
          inbound_schedd_port = 1966

cluster2: type=cluster
          outbound_hosts = M3(cluster1) M4(cluster3)
          inbound_hosts = M3(cluster1) M4(cluster3) M5(cluster3)
```

```
cluster3: type=cluster
          outbound_hosts = M6
          inbound_hosts = M6
```

Defining regions

The region stanza defines the managed region on the cluster for the region managers.

The region stanza is part of the **LoadL_admin** file and contains the **region_mgr_list** keyword. The **region_mgr_list** contains a list of machine names. The first entry is the primary region manager while the remainder are alternate region managers. The primary and alternate region managers will use the same failover scheme as the central manager.

Every machine listed in the **region_mgr_list** must be unique and must belong to the region.

A machine can belong to only one region. Every machine in a machine group belongs to the same region. A region cannot be specified in a machine substanza.

If required, a default region stanza must be specified explicitly. The default region is assigned to machines and machine groups that do not specify a region.

If there are any regions defined, then every machine or machine group must be assigned to a valid region either explicitly or by default.

The region manager daemon will only start if there is a region stanza defined. LoadLeveler will not start if the regions for the cluster are not defined correctly.

Note: The region manager node must have similar connectivity to the network as the executing machine it manages, so that all of its configured network interfaces are able to connect to all of the executing machine's network interfaces.

Region stanza format and keyword summary

Region stanzas take the following format.

```
label: type = region
region_mgr_list = list
```

Examples: Region stanzas

Any of these region stanza examples may apply to your situation.

- **Example 1**

  In this example, MachineGroupB is in the RegionB region. The primary machine to run the region manager daemon is c197blade4b06. c197blade4b11 is the backup or alternate node on which the region manager will come up if the primary node goes down.

  ```
  default: type = region
           region_mgr_list = c197blade4b22 c197blade4b24

  RegionA: type = region
           region_mgr_list = c197blade4b02 c197blade4b04

  RegionB: type = region
           region_mgr_list = c197blade4b06 c197blade4b11
  ```

```
MachineGroupA: type = machine_group
               machine_list = c197blade4b[01-05]
               region = RegionA
               ...

MachineGroupB: type = machine_group
               machine_list = c197blade4b[06-18]
               region = RegionB
               ...

MachineGroupC: type = machine_group
               machine_list = c197blade4b[20-24]
               region = default
               ...
```

- **Example 2**

  For an individual machine, the region keyword will specify the region to which the machine belongs. The region stanza will then show the primary region manager and the list of alternate region managers for that region.

```
RegionA: type = region
         region_mgr_list = c197blade4b02 c197blade4b04

c197blade4b02: type = machine
               region = RegionA
               ...

c197blade4b03: type = machine
               region = RegionA
               ...

c197blade4b04: type = machine
               region = RegionA
               ...
```

- **Example 3**

  This example shows that a default region stanza can be created so that the machine group and machine stanza can default to this region stanza if a region keyword is not specified in the machine group or machine stanza.

```
default: type = region
                region_mgr_list = c197blade4b22 c197blade4b24

# Machine group defaults to default region stanza
MachineGroupA: type = machine
               machine_list = c197blade4b[10-26]
               ...

# Machine defaults to default region stanza
c197blade4b05: type = machine
```

Chapter 6. Performing additional administrator tasks

There are additional ways to modify the LoadLeveler environment that either require an administrator to customize the configuration database or both the configuration and administration files, or require the use of the LoadLeveler commands or APIs.

Table 20 lists additional ways to modify the LoadLeveler environment.

Table 20. Roadmap of additional administrator tasks

To learn about:	Read the following:
Setting up the environment for parallel jobs	"Setting up the environment for parallel jobs" on page 110
Configuring and using an alternative scheduler	• "Using the BACKFILL scheduler" on page 114 • "Using an external scheduler" on page 119 • "Example: Changing scheduler types" on page 122
Using additional features available with the BACKFILL scheduler	• "Preempting and resuming jobs" on page 122 • "Configuring LoadLeveler to support reservations" on page 127 • "Working with reservations" on page 203 • "Data staging" on page 117 • "Configuring and using island scheduling" on page 165
Working with the workload balancing component	"Steps for integrating LoadLeveler with the Workload Manager" on page 133
Enabling LoadLeveler's checkpoint/restart function	"LoadLeveler support for checkpointing jobs" on page 135
Enabling LoadLeveler's checkpoint/restart function for jobs using MetaCluster HPC	"Submitting a MetaCluster HPC checkpoint job to LoadLeveler" on page 138
Enabling LoadLeveler's affinity support	• LoadLeveler scheduling affinity (see "LoadLeveler scheduling affinity support" on page 147)
Enabling LoadLeveler's multicluster support	• "LoadLeveler multicluster support" on page 149 • "Configuring a LoadLeveler multicluster" on page 150
Enabling LoadLeveler's Blue Gene support	• "LoadLeveler Blue Gene support" on page 153 • "Configuring LoadLeveler Blue Gene support" on page 155
Enabling LoadLeveler's fair share scheduling support	• "Fair share scheduling overview" on page 27 • "Using fair share scheduling" on page 158
Moving job records from a down Schedd to another Schedd within the local cluster	• "Procedure for recovering a job spool" on page 164 • **llmovespool** (see *LoadLeveler: Command and API Reference*)

Table 20. Roadmap of additional administrator tasks (continued)

To learn about:	Read the following:
Enabling energy aware job support	• "Energy aware job support" on page 166 • "S3 state support" on page 166 • "Working with energy aware jobs" on page 220
Correctly specifying configuration and administration file keywords	• Chapter 10, "Configuration keyword reference," on page 231 • Chapter 11, "Administration keyword reference," on page 293
Managing LoadLeveler operations	
• Querying status	• **llclass** • **llq** • **llqres** • **llstatus** See *LoadLeveler: Command and API Reference* for command descriptions.
• Changing attributes of submitted jobs	• **llfavorjob** • **llfavoruser** • **llmodify** • **llprio** See *LoadLeveler: Command and API Reference* for command descriptions.
• Changing the state of submitted jobs	• **llcancel** • **llhold** See *LoadLeveler: Command and API Reference* for command descriptions.

Setting up the environment for parallel jobs

Additional administration tasks apply to parallel jobs.

This topic describes the following administration tasks that apply to parallel jobs:

- Scheduling support
- Reducing job launch overhead
- Submitting interactive POE jobs
- Setting up a class
- Setting up a parallel master node

For information on submitting parallel jobs, see "Working with parallel jobs" on page 184.

Scheduling considerations for parallel jobs

For parallel jobs, LoadLeveler supports BACKFILL scheduling for efficient use of system resources. This scheduler runs both serial and parallel jobs.

BACKFILL scheduling also supports:

- Multiple tasks per node
- Multiple user space tasks per adapter
- Preemption

Specify the LoadLeveler scheduler using the **SCHEDULER_TYPE** keyword. For more information on this keyword and supported scheduler types, see "Choosing a scheduler" on page 46.

Steps for reducing job launch overhead for parallel jobs

Administrators may define a number of LoadLeveler starter processes to be ready and waiting to handle job requests. Having this pool of ready processes reduces the amount of time LoadLeveler needs to prepare jobs to run. You also may control how environment variables are copied for a job. Reducing the number of environment variables that LoadLeveler has to copy reduces the amount of time LoadLeveler needs to prepare jobs to run.

Before you begin: You need to know:

- How many jobs might be starting at the same time. This estimate determines how many starter processes to have LoadLeveler start in advance, to be ready and waiting for job requests.
- The type of parallel jobs that typically are used. If IBM Parallel Environment (PE) is used for parallel jobs, PE copies the user's environment to all executing nodes. In this case, you may configure LoadLeveler to avoid redundantly copying the same environment variables.
- How to correctly specify configuration keywords. For details about specific keyword syntax and use:
  - In the administration file, see Chapter 11, "Administration keyword reference," on page 293.
  - In the configuration file, see Chapter 10, "Configuration keyword reference," on page 231.

Perform the following steps to configure LoadLeveler to reduce job launch overhead for parallel jobs.

1. In the local or global configuration file, specify the number of starter processes for LoadLeveler to automatically start before job requests are submitted. Use the PRESTARTED_STARTERS keyword to set this value.

   Tip: The default value of 1 should be sufficient for most installations.
2. If typical parallel jobs use a facility such as Parallel Environment, which copies user environment variables to all executing nodes, set the **env_copy** keyword in the class, user, or group stanzas to specify that LoadLeveler only copy user environment variables to the master node by default.

   Rules:

   - Users also may set this keyword in the job command file. If the **env_copy** keyword is set in the job command file, that setting overrides any setting in the administration file. For more information, see "Step for controlling whether LoadLeveler copies environment variables to all executing nodes" on page 185.
   - If the **env_copy** keyword is set in more than one stanza in the administration file, LoadLeveler determines the setting to use by examining all values set in the applicable stanzas. See the table in the **env_copy** keyword in "Administration keyword descriptions" on page 298 to determine what value LoadLeveler will use.

3. Notify LoadLeveler daemons by issuing the **llctl** command with either the **reconfig** or **recycle** keyword. Otherwise, LoadLeveler will not process the modifications you made to the configuration and administration files.

When you are done with this procedure, you can use the POE stderr and LoadLeveler logs to trace actions during job launch.

Steps for allowing users to submit interactive POE jobs

Perform the following steps to set up your system so that users can submit interactive POE jobs to LoadLeveler.

1. Make sure that you have installed LoadLeveler and defined LoadLeveler administrators. See "Defining LoadLeveler administrators" on page 45 for information on defining LoadLeveler administrators.
2. If running user space jobs, the machine will need to be configured for InfiniBand switch adapters.
3. In the configuration file, define your scheduler to be the LoadLeveler BACKFILL scheduler by specifying **SCHEDULER_TYPE = BACKFILL**. See "Choosing a scheduler" on page 46 for more information.
4. In the administration file, specify batch, interactive, or general use for nodes. You can use the **machine_mode** keyword in the machine stanza to specify the type of jobs that can run on a node; you must specify either **interactive** or **general** if you are going to run interactive jobs.
5. In the administration file, configure optional functions, including:
   - Setting up pools: you can organize nodes into pools by using the **pool_list** keyword in the machine stanza. See "Defining machines" on page 89 for more information.
6. Consider setting up a class stanza for your interactive POE jobs. See "Setting up a class for parallel jobs" for more information. Define this class to be your default class for interactive jobs by specifying this class name on the **default_interactive_class** keyword. See "Defining users" on page 102 for more information.

Setting up a class for parallel jobs

To define the characteristics of parallel jobs run by your installation you should set up a class stanza in the administration file and define a class (in the **Class** statement in the configuration file) for each task you want to run on a node.

Suppose your installation plans to submit long-running parallel jobs, and you want to define the following characteristics:

- Only certain users can submit these jobs
- Jobs have a 30 hour run time limit
- A job can request a maximum of 60 nodes and 120 total tasks
- Jobs will have a relatively low run priority

The following is a sample class stanza for long-running parallel jobs which takes into account these characteristics:

```
long_parallel: type=class
wall_clock_limit = 1800
include_users = jack queen king ace
priority = 50
total_tasks = 120
max_node = 60
maxjobs = 2
```

Note the following about this class stanza:

- The **wall_clock_limit** keyword sets a wall clock limit of 1800 seconds (30 hours) for jobs in this class
- The **include_users** keyword allows four users to submit jobs in this class
- The **priority** keyword sets a relative priority of 50 for jobs in this class
- The **total_tasks** keyword specifies that a user can request up to 120 total tasks for a job in this class
- The **max_node** keyword specifies that a user can request up to 60 nodes for a job in this class
- The **maxjobs** keyword specifies that a maximum of two jobs in this class can run simultaneously

Suppose users need to submit job command files containing the following statements:

```
node = 30
tasks_per_node = 4
```

In your **LoadL_config** file, you must code the **Class** statement such that at least 30 nodes have four or more long_parallel classes defined. That is, the configuration file for each of these nodes must include the following statement:

```
Class = { "long_parallel" "long_parallel" "long_parallel" "long_parallel" }
```

or

```
Class = long_parallel(4)
```

For more information, see "Defining LoadLeveler machine characteristics" on page 59.

Striping when some networks fail

When multiple networks are configured in a cluster, a job can request striping over the networks by setting **sn_all** in the network statement in the job command file. The **striping_with_minimum_networks** administration file keyword in the class stanza is used to tell LoadLeveler how to select nodes for **sn_all** jobs of a specific class when one or more networks are unavailable. When **striping_with_minimum_networks** is set to **false** for a class, LoadLeveler will only select nodes for **sn_all** jobs of that class where all the networks are up and in the READY state. When **striping_with_minimum_networks** is set to **true**, LoadLeveler will select a set of nodes where at least more than half of the networks on the nodes are up and in the READY state.

For example, if there are 8 networks connected to a node and **striping_with_minimum_networks** is set to **false**, all 8 networks would have to be up and in the READY state to consider that node for **sn_all** jobs. If **striping_with_minimum_networks** is set to **true**, nodes with at least 5 networks up and in the READY state would be considered for **sn_all** jobs

Setting up a parallel master node

LoadLeveler allows you to define a parallel master node that LoadLeveler will use as the first node for a job submitted to a particular class. To set up a parallel master node, code the following keywords in the node's class and machine stanzas in the administration file:

```
# MACHINE STANZA: (optional)
mach1:     type = machine
master_node_exclusive = true


# CLASS STANZA: (optional)
pmv3:      type = class
master_node_requirement = true
```

Specifying **master_node_requirement = true** forces all parallel jobs in this class to use–as their first node–a machine with the **master_node_exclusive = true** setting. For more information on these keywords, see "Defining machines" on page 89 and "Defining classes" on page 94.

Using the BACKFILL scheduler

The BACKFILL scheduling algorithm in LoadLeveler is designed to maximize the use of resources to achieve the highest system efficiency, while preventing potentially excessive delays in starting jobs with large resource requirements.

These large jobs can run because the BACKFILL scheduler does not allow jobs with smaller resource requirements to continuously use up resources before the larger jobs can accumulate enough resources to run. While BACKFILL can be used for both serial and parallel jobs, the potential advantage is greater with parallel jobs.

Job steps are arranged in a queue based on their **SYSPRIO** order as they arrive from the Schedd nodes in the cluster. The queue can be periodically reordered depending on the value of the **RECALCULATE_SYSPRIO_INTERVAL** keyword. In each dispatching cycle, as determined by the **NEGOTIATOR_INTERVAL** and **NEGOTIATOR_CYCLE_DELAY** configuration keywords, the BACKFILL algorithm examines these job steps sequentially in an attempt to find available resources to run each job step, then dispatches those steps to run.

Once the BACKFILL algorithm encounters a job step for which it cannot immediately find enough resources, that job step becomes known as a "top dog". The BACKFILL algorithm can allocate multiple top dogs in the same dispatch cycle. By using the **MAX_TOP_DOGS** configuration keyword (for more information, see Chapter 10, "Configuration keyword reference," on page 231), you can define the maximum number of top dogs that the central manager will allocate. For each top dog, the BACKFILL algorithm will attempt to calculate the earliest time at which enough resources will become free to run the corresponding top dog. This is based on the assumption that each currently running job step will run until its hard wall clock limit is reached and that when a job step terminates, the resources which that step has been using will become available.

The time at which enough currently running job steps will have terminated, meaning enough resources have become available to run a top dog, is called top dog's future start time. The future start time of each top dog is effectively guaranteed for the remainder of the execution of the BACKFILL algorithm. All resources that each top dog will use at its corresponding start time and for its duration, as specified by its hard wall clock limit, are reserved (not to be confused with the reservation feature available in LoadLeveler). When reserving resources for a top dog, specific machines are reserved and if **RSET_MCM_AFFINITY** is specified then specific CPUs are reserved.

Note: A job that is bound to a reservation is not considered for top-dog scheduling, so there is no top-dog scheduling performed inside reservations.

In some cases, it may not be possible to calculate the future start time of a job step. Consider, for example, a case where there are 20 nodes in the cluster and a job step requires 24 nodes to run. Even when all nodes in the cluster are idle, it will not be possible for this job step to run. Only the addition of nodes to the cluster would allow the job step to run, and there is no way the BACKFILL algorithm can make any assumptions about when that could take place. In situations like this, the job step is not considered a "top dog", no resources are "reserved", and the BACKFILL algorithm goes on to the next job step in the queue.

The BACKFILL scheduling algorithm classifies job steps into distinct types: REGULAR, TOP DOG, and BACKFILL:

- The REGULAR job step is a job step for which enough resources are currently available and no top dogs have yet been allocated.
- The TOP DOG job step is a job step for which not enough resources are currently available, but enough resources are available at a future time and one of the following conditions is met:
  - The TOP DOG job step is not expected to run at a time when any other top dog is expected to run.
  - If the TOP DOG is expected to run at a time when some other top dogs are expected to run, then it cannot be using resources reserved by such top dogs.
- The BACKFILL job step is a job step for which enough resources are currently available and one of the following conditions is met:
  - The BACKFILL job step is expected to complete before the future start times of all top dogs, based on the hard wall clock limit of the BACKFILL job step.
  - If the BACKFILL job step is not expected to complete before the future start time of at least one top dog, then it cannot be using resources reserved by the top dogs that are expected to start before BACKFILL job step is expected to complete.

Table 21 provides a roadmap of BACKFILL scheduler tasks.

Table 21. Roadmap of BACKFILL scheduler tasks

Subtask	Associated instructions (see . . .)
Configuring the BACKFILL scheduler	• “Choosing a scheduler” on page 46 • “Tips for using the BACKFILL scheduler” on page 116 • “Example: BACKFILL scheduling” on page 117
Using additional LoadLeveler features available under the BACKFILL scheduler	• “Preempting and resuming jobs” on page 122 • “Configuring LoadLeveler to support reservations” on page 127 • “Working with reservations” on page 203 • “Data staging” on page 117

Table 21. Roadmap of BACKFILL scheduler tasks (continued)

Subtask	Associated instructions (see . . .)
Use the BACKFILL scheduler to dispatch and manage jobs	• **llclass** • **llmodify** • **llpreempt** • **llq** • **llsubmit** • Data access API • Error handling API • **ll_modify** • **ll_preempt** See *LoadLeveler: Command and API Reference* for command and API descriptions.

Tips for using the BACKFILL scheduler

Note the following when using the BACKFILL scheduler:

- To use this scheduler, either users must set a wall-clock limit in their job command file or the administrator must define a wall-clock limit value for the class to which a job is assigned. Jobs with the **wall_clock_limit** of **unlimited** cannot be used to backfill because they may not finish in time.
- Using wall clock limits that accurately reflect the actual running time of the job steps will result in a more efficient utilization of resources. When a job step's wall clock limit is substantially longer than the amount of time the job step actually needs, it results in two inefficiencies in the BACKFILL algorithm:
  - The future start time of a "top dog" will be calculated to be much later due to the long wall clock limits of the running job steps, leaving a larger window for BACKFILL job steps to run. This causes the "top dog" to start later than it would have if more accurate wall clock limits had been given.
  - A job step is less likely to be backfilled if its wall clock limit is longer because it is more likely to run past the future start time of a "top dog".
- You should use only the default settings for the **START** expression and the other job control functions described in “Managing job status through control expressions” on page 72. If you do not use these default settings, jobs will still run but the scheduler will not be as efficient. For example, the scheduler will not be able to guarantee a time at which the highest priority job will run.
- You should configure any multiprocessor (SMP) nodes such that the number of jobs that can run on a node (determined by the **MAX_STARTERS** keyword) is always less than or equal to the number of processors on the node.
- Due to the characteristics of the BACKFILL algorithm, in some cases this scheduler may not honor the **MACHPRIO** statement. For more information on **MACHPRIO**, see “Setting negotiator characteristics and policies” on page 47.
- When using **PREEMPT_CLASS** rules it is helpful to create a **SYSPRIO** expression which is consistent with the preemption rules. This can be done by using the **ClassSysprio** built-in variable with a multiplier, such as `SYSPRIO: (ClassSysprio * 10000) - QDate`. If classes which appear on the left-hand side of **PREEMPT_CLASS** rules are given a higher priority than those which appear on the right, preemption won't be required as often because the job steps which can preempt will be higher in the queue than the job steps which can be preempted.

- Entering **llq -s** against a top-dog step will display that this step is a top-dog.

Example: BACKFILL scheduling

On a rack with 10 nodes, 8 of the nodes are being used by Job A. Job B has the highest priority in the queue, and requires 10 nodes. Job C has the next highest priority in the queue, and requires only two nodes. Job B has to wait for Job A to finish so that it can use the freed nodes. Because Job A is only using 8 of the 10 nodes, the BACKFILL scheduler can schedule Job C (which only needs the two available nodes) to run as long as it finishes before Job A finishes (and Job B starts). To determine whether or not Job C has time to run, the BACKFILL scheduler uses Job C's **wall_clock_limit** value to determine whether or not it will finish before Job A ends. If Job C has a **wall_clock_limit** of **unlimited**, it may not finish before Job B's start time, and it won't be dispatched.

Data staging

Data staging allows you to stage data needed by a job before the job begins execution and to move data back to archives when a job has finished execution. A job can use one inbound data staging step and one outbound data staging step. The inbound step will be the first to be executed and the outbound step, the last.

LoadLeveler provides data staging for two scenarios:

1. A single replica of the data files needed by a job have to be created on a common file system.
2. A replica of the data files has to be created on every machine on which the job will run.

LoadLeveler allows you to request the time at which data staging operations should be scheduled.

1. A single replica must be created as soon as a job is submitted, regardless of when the job will be executed. This is the **AT_SUBMIT** configuration option.
2. A single replica of the data files must be created as close as possible to execution time of the job. This is the **JUST_IN_TIME** configuration option.
3. A replica must be created on each machine that the job runs on, as close as possible to execution time of the job. This is also the **JUST_IN_TIME** configuration option.

The basic steps involved in data staging include:

1. A job is submitted that contains data staging keywords.
2. LoadLeveler generates inbound and outbound data staging steps in accordance with these keywords. All other steps of the job have an implicit dependency on the completion of the inbound data staging step.
3. Scheduling methods:
   a. With the **AT_SUBMIT** configuration option, the data staging step is started first and the application steps are scheduled when its data staging dependency is satisfied (that is, when the inbound data staging step is completed).
   b. With the **JUST_IN_TIME** configuration option, the first application step of the job is scheduled in the future based on the wall clock time specified for the inbound data staging step. The inbound data staging step is started on the machines that will be used by the first application step.
4. When the inbound data staging step completes, all of the application job steps become eligible for scheduling. The exit code from the inbound data staging

program is made available to all application job steps in the LL_DSTG_IN_EXIT_CODE environment variable.

5. When all the application job steps are completed, the outbound data staging step is started by LoadLeveler. Typically, the outbound data staging step would be used to move data files back to their archives.

Note: You cannot preempt data staging steps using the **llpreempt** command or by specifying the **data_stage** class in system preemption rules. Similarly, a step belonging to the **data_stage** class cannot preempt any other job step.

Configuring LoadLeveler to support data staging

LoadLeveler allows you to specify the execution time for data staging job steps using the **DSTG_TIME** keyword. It defaults to the **AT_SUBMIT** value. To schedule data staging operation as close to the application as possible, the **JUST_IN_TIME** value can be used. **DSTG_MIN_SCHEDULING_INTERVAL** is a keyword used to optimize scheduler performance by allowing data staging jobs to be scheduled only at specific intervals.

A special set of data staging step initiators, called **DSTG_MAX_STARTERS**, can be set up for data staging job steps. These initiators will be a distinct set of resources on the executing machine, not included in the **MAX_STARTERS** set up for compute jobs. You cannot specify the built-in **data_stage** class in:

- The **CLASS** keyword of a job command file
- The **default_class** keyword in the administration file

For more information about the data staging keywords, see "Configuration keyword descriptions" on page 233.

The LoadLeveler administration class stanza keywords can be used to specify defaults, limits, and restrictions for the built-in **data_stage** class. The **data_stage** class cannot be specified as the default class for a user. You cannot specify the **data_stage** class in your job command file. Steps of this class will be automatically generated by LoadLeveler based on the data staging keywords used in job command files.

LoadLeveler provides a built-in class called **data_stage** that can be configured in the administration file using a class stanza, just as you would do for any other class. Some examples of how you might use a stanza for the **data_stage** class are:

- Include and exclude users and groups from this class to control which users are permitted to use data staging.
- Specifying defaults for resource limits such as **cpu_limit** or **nofile_limit** for data staging steps.
- Specifying defaults and maximum allowed values for the **dstg_resources** job command file keyword using **default_resources** and **max_resources**.
- Limiting the total number of data staging jobs or tasks in the cluster at any one time using **maxjobs** or **max_total_tasks**.

For more information about the data staging keywords, see "Administration keyword descriptions" on page 298.

If an inbound data staging job step is soft-bound to a reservation and keyword **dstg_node=any**, it can be started ahead of the reservation start time, if data staging resources are available. In all other cases, data staging steps will run within the reservation itself.

Using an external scheduler

The LoadLeveler API provides interfaces that allow an external scheduler to manage the assignment of resources to jobs and dispatching those jobs.

The primary interfaces for the tasks of an external scheduler are:

- **ll_query** to obtain information about the LoadLeveler cluster, the machines of the cluster, jobs and Workload Manager.
- **ll_get_data** to obtain information about specific objects such as jobs, machines and adapters.
- **ll_start_job_ext** to start a LoadLeveler job.
  - The **ll_start_job_ext** subroutine supports both serial and parallel jobs. For parallel jobs, **ll_start_job_ext** provides the ability to specify which adapters are used by the communication protocols of each job task. This assures that each task uses the same network for communication over a given protocol.

The steps for dispatching jobs with an external scheduler are:

1. Gather information about the LoadLeveler cluster **(ll_query(CLUSTER)).**
2. Gather information about the machines in the LoadLeveler cluster **(ll_query(MACHINES)).**
3. Gather information about the jobs in the cluster **(ll_query(JOBS)).**
4. Determine the resources that are currently free. (See the note that follows.)
5. Determine which jobs to start. Assign resources to jobs to be started and dispatch **(ll_start_job_ext(LL_start_job_info_ext*))**.
6. Repeat steps 1 through 5.

When an external scheduler is used, the LoadLeveler Negotiator does not keep track of the resources used by jobs started by the external scheduler. There are two ways that an external scheduler can keep track of the free resources available for starting new jobs. The method that should be used depends on whether the external scheduler runs continuously while all scheduling is occurring or is executed to start a finite number of jobs and then terminates:

- If the external scheduler runs continuously, it should query the total resources available in the LoadLeveler system with **ll_query** and **ll_get_data**. Then it can keep track of the resource assigned to jobs it starts while they are running and return the resources to the available pool when the jobs complete.
- If the external scheduler is executed to start a finite number of jobs and then terminates, it must determine the pool of available resources when it first starts. It can do this by first querying the total resources in the LoadLeveler system using **ll_query** and **ll_get_data**. Then it would query the jobs in the system (again using **ll_query**), looking for jobs that are running. For each running job, it would remove the resources used by the job from the available pool. After all the running jobs are processed, the available pool would indicate the amount of free resource for starting new jobs.

To find out more about dispatching jobs with an external scheduler, use the information in Table 22 on page 120.

Table 22. Roadmap of tasks for using an external scheduler

Subtask	Associated instructions (see . . .)
Learn about the LoadLeveler functions that are limited or not available when you use an external scheduler	"Replacing the default LoadLeveler scheduling algorithm with an external scheduler"
Prepare the LoadLeveler environment for using an external scheduler	"Customizing the configuration file to define an external scheduler" on page 121

Replacing the default LoadLeveler scheduling algorithm with an external scheduler

It is important to know how LoadLeveler keywords and commands behave when you replace the default LoadLeveler scheduling algorithm with an external scheduler. LoadLeveler scheduling keywords and commands fall into the following categories:

- Keywords not involved in scheduling decisions are unchanged.
- Keywords kept in the job object or in the machine which are used by the LoadLeveler default scheduler have their values maintained as before and passed to the data access API.
- Keywords used only by the LoadLeveler default scheduler have no effect.

Table 23 discusses specific keywords and commands and how they behave when you disable the default LoadLeveler scheduling algorithm.

Table 23. Effect of LoadLeveler keywords under an external scheduler

Keyword type / name	Notes
Job command file keywords	
class	This value is provided by the data access API. Machines chosen by **ll_start_job_ext** *must* have the class of the job available or the request will be rejected.
dependency	Supported as before. Job objects for which dependency cannot be evaluated (because a previous step has not run) are maintained in the NotQueued state, and attempts to start them using **ll_start_job_ext** will result in an error. If the dependency is met, **ll_start_job_ext** can start the proc.
hold	**ll_start_job_ext** cannot start a job that is in Hold status.
preferences	Passed to the data access API.
requirements	**ll_start_job_ext** returns an error if the specified machines do not match the requirements of the job. This includes Disk and Virtual Memory requirements.
startdate	The job remains in the Deferred state until the **startdate** specified in the job is reached. **ll_start_job_ext** cannot start a job in the Deferred state.

Table 23. Effect of LoadLeveler keywords under an external scheduler (continued)

Keyword type / name	Notes
user_priority	Used in calculating the system priority (as described in "Setting and changing the priority of a job" on page 224). The system priority assigned to the job is available through the data access API. No other control of the order in which jobs are run is enforced.
Administration file keywords	
master_node_exclusive	Ignored
master_node_requirement	Ignored
max_jobs_scheduled	Ignored
max_reservations	Ignored
max_reservation_duration	Ignored
max_total_tasks	Ignored
maxidle	Supported
maxjobs	Ignored
maxqueued	Supported
priority	Used to calculate the system priority (where appropriate).
speed	Available through the data access API.
Configuration file keywords	
MACHPRIO	Calculated but is not used.
MAX_STARTERS	Calculated, and if starting the job causes this value to be exceeded, **ll_start_job_ext** returns an error.
SYSPRIO	Calculated and available to the data access API.
NEGOTIATOR_PARALLEL_DEFER	Ignored
NEGOTIATOR_PARALLEL_HOLD	Ignored
NEGOTIATOR_RESCAN_QUEUE	Ignored
NEGOTIATOR_RECALCULATE_ SYSPRIO_INTERVAL	Works as before. Set this value to 0 if you do not want the system priorities of job objects recalculated.

Customizing the configuration file to define an external scheduler

To use an external scheduler, one of the tasks you must perform is setting the configuration file keyword **SCHEDULER_TYPE** to the value **API**. This keyword option provides a time-based (rather than an event-based) interface. That is, your application must use the data access API to poll LoadLeveler at specific times for machine and job information.

When you enable a scheduler type of API, you must specify **AGGREGATE_ADAPTERS=NO** to make the individual switch adapters available to the external scheduler. This means the external scheduler receives each individual adapter connected to the network, instead of collectively grouping them together. You'll see each adapter listed individually in the **llstatus -l** command output. When this keyword is set to **YES**, the **llstatus -l** command will show an

aggregate adapter which contains information on all switch adapters on the same network. For detailed information about individual switch adapters, issue the **llstatus -a** command.

You also may use the **PREEMPTION_SUPPORT** keyword, which specifies the level of preemption support for a cluster. Preemption allows for a running job step to be suspended so that another job step can run.

Example: Retrieving specific information

For a sample that demonstrates retrieving information about the LoadLeveler cluster, its machines, and jobs, assigning resources, and dispatching jobs see the LoadLeveler samples directory:

- On AIX:

  /usr/lpp/LoadL/scheduler/full/samples/llsch
- On Linux:

  /opt/ibmll/LoadL/scheduler/full/samples/llsch

Example: Changing scheduler types

You can toggle between the default LoadLeveler scheduler and other types of schedulers by using the **SCHEDULER_TYPE** keyword.

Changes to **SCHEDULER_TYPE** will not take effect at reconfiguration. The administrator must stop and restart or recycle LoadLeveler when changing **SCHEDULER_TYPE**. A combination of changes to **SCHEDULER_TYPE** and some other keywords may terminate LoadLeveler.

The following example illustrates how you can toggle between the default LoadLeveler scheduler and an external scheduler, such as the Extensible Argonne Scheduling sYstem (EASY), developed by Argonne National Laboratory and available as public domain code.

If you are running the default LoadLeveler scheduler, perform the following steps to switch to an external scheduler:

1. In the configuration file, set **SCHEDULER_TYPE = API**
2. On the central manager machine:
   - Issue **llctl -g stop** and **llctl -g start**, or
   - Issue **llctl -g recycle**

If you are running an external scheduler, this is how you can re-enable the LoadLeveler scheduling algorithm:

1. In the configuration file, set **SCHEDULER_TYPE = LL_DEFAULT**
2. On the central manager machine:
   - Issue **llctl -g stop** and **llctl -g start**, or
   - Issue **llctl -g recycle**

Preempting and resuming jobs

The BACKFILL scheduler allows LoadLeveler jobs to be preempted so that a higher priority job step can run.

Administrators may specify not only preemption rules for job classes, but also the method that LoadLeveler uses to preempt jobs. The BACKFILL scheduler supports various methods of preemption.

Use Table 24 to find more information about preemption.

Table 24. Roadmap of tasks for using preemption

Subtask	Associated instructions (see . . .)
Learn about types of preemption and what it means for preempted jobs	"Overview of preemption"
Prepare the LoadLeveler environment and jobs for preemption	"Planning to preempt jobs" on page 124
Configure LoadLeveler to use preemption	"Steps for configuring a scheduler to preempt jobs" on page 126

Overview of preemption

LoadLeveler supports two types of preemption:

- System-initiated preemption
  - Automatically enforced by LoadLeveler, except for job steps running under a reservation.
  - Governed by the PREEMPT_CLASS rules defined in the global configuration file.
  - When resources required by an incoming job are in use by other job steps, all or some of those job steps in certain classes may be preempted according to the PREEMPT_CLASS rules.
  - An automatically preempted job step will be resumed by LoadLeveler when resources become available and conditions such as START_CLASS rules are satisfied.
  - An automatically preempted job step cannot be resumed using **llpreempt** command or **ll_preempt** subroutine.
- User-initiated preemption
  - Manually initiated by LoadLeveler administrators using **llpreempt** command or **ll_preempt** subroutine.
  - A manually preempted job step cannot be resumed automatically by LoadLeveler.
  - A manually preempted job step can be resumed using **llpreempt** command or **ll_preempt** subroutine. Issuing this command or subroutine, however, does not guarantee that the job step will successfully be resumed. A manually preempted job step that was resumed through these interfaces competes for resources with system-preempted job steps, and will be resumed only when resources become available.
  - All steps in a set of coscheduled job steps will be preempted if one or more steps in the step is preempted.
  - A coscheduled step will not be resumed until all steps in the set of coscheduled job steps can be resumed.

For the BACKFILL scheduler only, administrators may select which method LoadLeveler uses to preempt and resume jobs. The suspend method is the default behavior, and is the preemption method LoadLeveler uses for any external schedulers that support preemption. For more information about preemption methods, see "Planning to preempt jobs" on page 124.

Note: An MPICH2 job cannot be preempted with the suspend method (the job state will change to E (preempted), but the children processes will still be running).

For a preempted job to be resumed after system- or user-initiated preemption occurs through a method other than suspend, the **restart** keyword in the job command file must be set to **yes**. Otherwise, LoadLeveler vacates the job step and removes it from the cluster.

In order to determine the preempt type and preempt method to use when a coscheduled step preempts another step, an order of precedence for preempt types and preempt methods has been defined. All steps in the preempting coscheduled step will be examined and the preempt type and preempt method having the highest precedence will be used. The order of precedence for preempt type will be ALL, ENOUGH. The precedence order for preempt method will be remove, vacate, system hold, user hold, suspend.

When coscheduled steps are running, if one step is preempted as a result of a system initiated preemption, then all coscheduled steps will be preempted. This implies that more resource than necessary might be preempted when one of the steps being preempted is a coscheduled step.

Planning to preempt jobs

Consider the following points when planning to use preemption:

- **Avoiding circular preemption under the BACKFILL scheduler**

  BACKFILL scheduling enables job preemption using rules specified with the **PREEMPT_CLASS** keyword. When you are setting up the preemption rules, make sure that you do not create a circular preemption path. Circular preemption causes a job class to preempt itself after applying the preemption rules recursively. For example, the following keyword definitions set up circular preemption rules on Class_A:

  ```
  PREEMPT_CLASS[Class_A] = ALL { Class_B }
  PREEMPT_CLASS[Class_B] = ALL { Class_C }
  PREEMPT_CLASS[Class_C] = ENOUGH { Class_A }
  ```

  Another example of circular preemption involves **allclasses**:

  ```
  PREEMPT_CLASS[Class_A] = ENOUGH {allclasses}
  PREEMPT_CLASS[Class_B] = ALL {Class_A}
  ```

  In this instance, **allclasses** means all classes except Class_A, any additional preemption rule preempting Class_A causes circular preemption.

- **Understanding implied START_CLASS values**

  Using the "ALL" value in the **PREEMPT_CLASS** keyword places implied restrictions on when a job can start. For example,

  ```
  PREEMPT_CLASS[Class_A] = ALL {Class_B Class_C}
  ```

  tells LoadLeveler two things:

  1. If a new Class_A job is about to run on a node set, then preempt all Class_B and Class_C jobs on those nodes
  2. If a Class_A job is running on a node set, then do not start any Class_B or Class_C jobs on those nodes

  This **PREEMPT_CLASS** statement also implies the following **START_CLASS** expressions:

  1. START_CLASS[Class_B] = (Class_A < 1)
  2. START_CLASS[Class_C] = (Class_A < 1)

LoadLeveler adds all implied **START_CLASS** expressions to the **START_CLASS** expressions specified in the configuration file. This overrides any existing values for **START_CLASS**.

For example, if the configuration file contains the following statements:

```
PREEMPT_CLASS[Class_A] = ALL {Class_B Class_C}
START_CLASS[Class_B] = (Class_A < 5)
START_CLASS[Class_C] = (Class_C < 3)
```

When LoadLeveler runs through the configuration process, the **PREEMPT_CLASS** statement on the first line generates the two implied **START_CLASS** statements. When the implied **START_CLASS** statements get added in, the user specified **START_CLASS** statements are overridden and the resulting **START_CLASS** statements are effectively equivalent to:

```
START_CLASS[Class_B] = (Class_A < 1)
START_CLASS[Class_C] = (Class_C < 3) && (Class_A < 1)
```

Note: LoadLeveler's central manager (CM) uses these effective expressions instead of the original statements specified in the configuration file. The output from **llclass -l** displays the original customer specified **START_CLASS** expressions.

- **Selecting the preemption method under the BACKFILL scheduler**

  Use Table 25 and Table 26 on page 126 to determine which preemption you want to use for jobs running under the BACKFILL scheduler. You may define one or more of the following:
  - A default preemption method to be used for all job classes, by setting the **DEFAULT_PREEMPT_METHOD** keyword in the configuration file.
  - A specific preemption method for one or more classes or job steps, by using an option on:
    - The **PREEMPT_CLASS** statement in the configuration file.
    - The **llpreempt** command, **ll_preempt** subroutine or **ll_preempt_jobs** subroutine.

  Notes:

  1. Process tracking must be enabled in order to use the suspend method to preempt a job. To configure LoadLeveler for process tracking, see "Tracking job processes" on page 73.
  2. For a preempted job to be resumed after system- or user-initiated preemption occurs through a method other than suspend and remove, the **restart** keyword in the job command file must be **set** to yes. Otherwise, LoadLeveler vacates the job step and removes it from the cluster.

Table 25. Preemption methods for which LoadLeveler automatically resumes preempted jobs

Preemption method (abbreviation)	**LoadLeveler resumes preempted job:**		
	At this time	**At this location**	**At this processing point**
Suspend (su)	When preempting job completes	On the same nodes	At the point of suspension
Vacate (vc)	When nodes are available	Any nodes that meet job requirements	At the beginning or at the last successful checkpoint

Table 26. Preemption methods for which administrator or user intervention is required

Preemption method (abbreviation)	Required intervention	LoadLeveler resumes preempted job:	
		At this location	At this processing point
Remove (rm)	Administrator or user must resubmit the preempted job	Any nodes that meet job requirements, when they are available	At the beginning or at the last successful checkpoint
System Hold (sh)	Administrator must release the preempted job		
User Hold (uh)	User must release the preempted job		

- **Understanding how LoadLeveler treats resources held by jobs to be preempted**

  When a job step is running, it may be holding the following resources:
  - Processors
  - Scheduling slots
  - Real memory
  - ConsumableCpus, ConsumableMemory, ConsumableVirtualMemory, and ConsumableLargePageMemory
  - Switch adapter windows

  When LoadLeveler suspends preemptable jobs running under the BACKFILL scheduler, certain resources held by those jobs do not become available for the preempting jobs. These resources include ConsumableVirtualMemory, ConsumableLargePageMemory, and floating resources. Under the BACKFILL scheduler only, LoadLeveler releases these resources when you select a preemption method other than suspend. For all preemption methods other than suspend, LoadLeveler treats all job-step resources as available when it preempts the job step.

  There is a special way in which LoadLeveler treats adapter window resources during preemption by suspend. If the **PREEMPTION_SUPPORT** is not set to full in the LoadLeveler configuration, the adapter resources are not released. If it is set to full, the adapter window resources of the preempted job are released, but are only made available to the preempting job. These resources are treated as in-use when scheduling jobs that did not preempt the preempted job.

- **Understanding how LoadLeveler processes multiple entries for the same keywords**

  If there are multiple entries for the same keyword in either a configuration file or an administration file, the last entry wins. For example, the following statements are all valid specifications for the same keyword **START_CLASS**:

  ```
  START_CLASS [Class_B] = (Class_A < 1)
  START_CLASS [Class_B] = (Class_B < 1)
  START_CLASS [Class_B] = (Class_C < 1)
  ```

  However, all three statements identify Class_B as the incoming class. LoadLeveler resolves these statements according to the "last one wins" rule. Because of that, the actual value used for the keyword is (Class_C < 1).

Steps for configuring a scheduler to preempt jobs

You need to know certain details about the job characteristics and workload at your installation before you begin to define rules for starting and preempting jobs.

Before you begin:

- To define rules for starting and preempting jobs, you need to know certain details about the job characteristics and workload at your installation, including:
  - Which jobs require the same resources, or must be run on the same machines, and so on. This knowledge allows you to group specific jobs into a class.
  - Which jobs or classes have higher priority than others. This knowledge allows you to define which job classes can preempt other classes.
- To correctly configure LoadLeveler to preempt jobs, you might need to refer to the following information:
  - "Choosing a scheduler" on page 46.
  - "Planning to preempt jobs" on page 124.
  - Chapter 10, "Configuration keyword reference," on page 231.
  - Chapter 11, "Administration keyword reference," on page 293.
  - **llctl** command (see *LoadLeveler: Command and API Reference*)

Perform the following steps to configure a scheduler to preempt jobs:

1. In the configuration file, use the **SCHEDULER_TYPE** keyword to define the type of LoadLeveler or external scheduler you want to use. Of the LoadLeveler schedulers, only the BACKFILL scheduler supports preemption.

   Rule: If you select the BACKFILL or API scheduler, you must set the **PREEMPTION_SUPPORT** configuration keyword to either **full** or **no_adapter**.
2. (Optional) In the configuration file, use the **DEFAULT_PREEMPT_METHOD** to define the default method that the BACKFILL scheduler should use for preempting jobs.

   Alternative: You also may set the preemption method through the **PREEMPT_CLASS** keyword or on the LoadLeveler preemption command or APIs, which override the setting for the **DEFAULT_PREEMPT_METHOD** keyword.
3. For either the BACKFILL or API scheduler, to preempt by the suspend method requires that you set the **PROCESS_TRACKING** configuration keyword to **true**.
4. In the configuration file, use the **PREEMPT_CLASS** and **START_CLASS** to define the preemption and start policies for job classes.
5. In the administration file, use the **max_total_tasks** keyword to define the maximum number of tasks that may be run per user, group, or class.
6. On the central manager machine:
   - Issue **llctl -g stop** and **llctl -g start**, or
   - Issue **llctl -g recycle**

When you are done with this procedure, you can use the **llq** command to determine whether jobs are being preempted and resumed correctly. If not, use the LoadLeveler logs to trace the actions of each daemon involved in preemption to determine the problem.

Configuring LoadLeveler to support reservations

Under the BACKFILL scheduler only, LoadLeveler allows authorized users to make reservations or recurring reservations, which specify one or more time periods during which specific node resources are reserved for use by particular users or groups.

Normally, jobs wait to be dispatched until the resources they require become available. Through the use of reservations, wait time can be reduced because only jobs that are bound to the reservation may use the node resources as soon as the reservation period begins.

Reservation tasks for administrators

Use Table 27 to find additional information about reservations.

Table 27. Roadmap of reservation tasks for administrators

Subtask	Associated instructions (see . . .)
Learn how reservations work in the LoadLeveler environment	• "Overview of reservations" on page 24 • "Understanding the reservation life cycle" on page 205
Configuring a LoadLeveler cluster to support reservations	• "Steps for configuring reservations in a LoadLeveler cluster" • "Examples: Reservation keyword combinations in the administration file" on page 130 • "Collecting accounting data for reservations" on page 67
Working with reservations: • Creating reservations • Submitting jobs under a reservation • Managing reservations	"Working with reservations" on page 203
Correctly coding and using administration and configuration keywords	• Chapter 11, "Administration keyword reference," on page 293 • Chapter 10, "Configuration keyword reference," on page 231
Failover and shutdown procedures	See the *TWS LoadLeveler for AIX: Installation Guide* or the *LoadLeveler for Linux: Installation Guide*.

Steps for configuring reservations in a LoadLeveler cluster

Only the BACKFILL scheduler supports the use of reservations.

Before you begin:

- For information about configuring the BACKFILL scheduler, see "Choosing a scheduler" on page 46.
- You need to decide:
  - Which users will be allowed to create reservations.
  - How many reservations users may own, and how long a duration for their reservations will be allowed.
  - Which nodes will be used for reservations.
  - How much setup time is required before the reservation period starts.
  - Whether accounting data for reservations is to be saved.
  - The maximum lifetime for a recurring reservation before you require the user to request a new reservation for that job.
  - Additional system-wide limitations that you may want to implement such as maintenance time blocks for specific node sets.
- For examples of possible reservation keyword combinations, see "Examples: Reservation keyword combinations in the administration file" on page 130.

- For details about specific keyword syntax and use:
  - In the administration file, see Chapter 11, "Administration keyword reference," on page 293.
  - In the configuration file, see Chapter 10, "Configuration keyword reference," on page 231.

Perform the following steps to configure reservations:

1. In the administration file, modify the user or group stanzas to authorize users to create reservations. You may grant the ability to create reservations to an individual user, a group of users, or a combination of users and groups. To do so, define the following keywords in the appropriate user or group stanzas:
   - **max_reservations**, to set the maximum number of reservations that a user or group may have.
   - (Optional) **max_reservation_duration**, to set the maximum amount of time for the reservation period.

   Tip: To quickly set up and use reservations, use one of the following examples:
   - To allow every user to create a reservation, add `max_reservations=1` to the default user stanza. Then every administrator or user may create a reservation, as long as the number of reservations has not reached the limit for a LoadLeveler cluster.
   - To allow a specific group of users to make 10 reservations, add `max_reservations=10` to the group stanza for that LoadLeveler group. Then every user in that group may create a reservation, as long as the number of reservations has not reached the limit for that group or for a LoadLeveler cluster.

   See the **max_reservations** description in Chapter 11, "Administration keyword reference," on page 293 for more information about setting this keyword in the user or group stanza.
2. In the administration file, modify the machine stanza of each machine that may be reserved. To do so, set the **reservation_permitted** keyword to **true**.

   Tip: If you want to allow every machine to be reserved, you do not have to set this keyword; by default, any LoadLeveler machine may be reserved. If you want to prevent particular machines from being reserved, however, you must define a machine stanza for that machine and set the **reservation_permitted** keyword to **false**.
3. In the global configuration file, set reservation policy by specifying values for the following keywords:
   - **MAX_RESERVATIONS** to specify the maximum number of reservations per cluster.

     Note: A recurring reservation only counts as one reservation towards the **MAX_RESERVATIONS** limit regardless of the number of times that the reservation recurs.
   - **RESERVATION_CAN_BE_EXCEEDED** to specify whether LoadLeveler will be permitted to schedule job steps bound to a reservation when their expected end times exceed the reservation end time.

     The default for this keyword is `TRUE`, which means that LoadLeveler will schedule these bound job steps even when they are expected to continue running beyond the time at which the reservation ends. Whether these job steps run and successfully complete depends on resource availability, which is not guaranteed after the reservation ends. In addition, these job steps become subject to preemption rules after the reservation ends.

Tip: You might want to set this keyword value to `FALSE` to prevent users from binding long-running jobs to run under reservations of short duration.

- **RESERVATION_MIN_ADVANCE_TIME** to define the minimum time between the time at which a reservation is created and the time at which the reservation is to start.

  Tip: To reduce the impact to the currently running workload, consider changing the default for this keyword, which allows reservations to begin as soon as they are created. You may, for example, require reservations to be made at least one day (1440 minutes) in advance, by specifying `RESERVATION_MIN_ADVANCE_TIME=1440` in the global configuration file.
- **RESERVATION_PRIORITY** to define whether LoadLeveler administrators may reserve nodes on which running jobs are expected to end after the start time for the reservation.

  Tip: The default for this keyword is `NONE`, which means that LoadLeveler will not reserve a node on which running jobs are expected to end after the start time for the reservation. If you want to allow LoadLeveler administrators to reserve specific nodes regardless of the expected end times of job steps currently running on the node, set this keyword value to `HIGH`. Note, however, that setting this keyword value to `HIGH` might increase the number of job steps that must be preempted when LoadLeveler sets up the reservation, and many jobs might remain in Preempted state. This also applies to Blue Gene job steps.

  This keyword value applies only for LoadLeveler administrators; other reservation owners do not have this capability.
- **RESERVATION_SETUP_TIME** to define the amount of time LoadLeveler uses to prepare for a reservation before it is to start.

4. (Optional) In the global configuration file, set controls for the collection of accounting data for reservations:
   - To turn on accounting for reservations, add the **A_RES** flag to the **ACCT** keyword.
   - To specify a file other than the default history file to contain the data, use the **RESERVATION_HISTORY** keyword.

   To learn how to collect accounting data for reservations, see "Collecting accounting data for reservations" on page 67.
5. If LoadLeveler is already started, to process the changes you made in the preceding steps, issue the command **llctl -g reconfig**.

   Tip: If you have changed the value of only the **RESERVATION_PRIORITY** keyword, issue the command **llctl reconfig** only on the central manager node.

   Result: The new keyword values take effect immediately, but they do not change the attributes of existing reservations.

When you are done with this procedure, you may perform additional tasks described in "Working with reservations" on page 203.

Examples: Reservation keyword combinations in the administration file

The following examples demonstrate LoadLeveler behavior when the **max_reservations** and **max_reservation_duration** keywords are set. The examples assume that only the user and group stanzas listed exist in the LoadLeveler administration file.

- **Example 1:** Assume the administration file contains the following stanzas:

```
default: type = user
         maxjobs = 10

group2: type = group
        include_users = rich dave steve

rich: type = user
      default_group = group2
```

This example shows that, by default, no one is allowed to make any reservations. No one, including LoadLeveler administrators, is permitted to make any reservations unless the **max_reservations** keyword is used.

- **Example 2:** Assume the administration file contains the following stanzas:

```
default: type = user
         maxjobs = 10

group2: type = group
        include_users = rich dave steve

rich: type = user
      default_group = group2
      max_reservations = 5
```

This example shows how permission to make reservations can be granted to a specific user through the user stanza only. Because the **max_reservations** keyword is not used in any group stanza, by default, the group stanzas neither grant permissions nor put any restrictions on reservation permissions. User Rich can make reservations in any group (group2, No_Group, Group_A, and so on), whether or not the group stanzas exist in the LoadLeveler administration file. The total number of reservations user Rich can own at any given time is limited to five.

- **Example 3:** Assume the administration file contains the following stanzas:

```
default: type = user
         maxjobs = 10

group2: type = group
        include_users = rich dave steve
        max_reservations = 5

rich: type = user
      default_group = group2
```

This example shows how permission to make reservations can be granted to a group of users through the group stanza only. Because the **max_reservations** keyword is not used in any user stanza, by default, the user stanzas neither grant nor deny permission to make reservations. All users in group2 (Rich, Dave and Steve) can make reservations, but they must make reservations in group2 because other groups do not grant the permission to make reservations. The total number of reservations the users in group2 can own at any given time is limited to five.

- **Example 4:** Assume the administration file contains the following stanzas:

```
default: type = user
         maxjobs = 10

group2: type = group
        include_users = rich dave steve
        max_reservations = 5

rich: type = user
      default_group = group2
      max_reservations = 0
```

This example shows how permission to make reservations can be granted to a group of users except one specific user. Because the **max_reservations** keyword is set to zero in the user stanza for Rich, he does not have permission to make any reservation, even though all other users in group2 (Dave and Steve) can make reservations.

- **Example 5:** Assume the administration file contains the following stanzas:

```
default: type = group
         max_reservations = 0

default: type = user
         max_reservations = 0

group2: type = group
        include_users = rich dave steve
        max_reservations = 5

rich: type = user
      default_group = group2
      max_reservations = 5

dave: type = user
      max_reservations = 2
```

  This example shows how permission to make reservations can be granted to specific user and group pairs. Because the **max_reservations** keyword is set to zero in both the default user and group stanza, no one has permission to make any reservation unless they are specifically granted permission through both the user and group stanza. In this example:
  - User Rich can own at any time up to five reservations in group2 only.
  - User Dave can own at any time up to two reservations in group2 only.

  The total number of reservations they can own at any given time is limited to five. No other combination of user or group pairs can make any reservations.

- **Example 6:** Assume the administration file contains the following stanzas:

```
default: type = user
         max_reservations = 1
```

  This example permits any user to make one reservation in any group, until the number of reservations reaches the maximum number allowed in the LoadLeveler cluster.

- **Example 7:** Assume the administration file contains the following stanzas:

```
default: type = group
         max_reservations = 0

default: type = user
         max_reservations = 0

group1: type = group
        max_reservations = 6
        max_reservation_duration = 1440

carol: type = user
       default_group = group1
       max_reservations = 4
       max_reservation_duration = 720

dave: type = user
      default_group = group1
      max_reservations = 4
      max_reservation_duration = 2880
```

  In this example, two users, Carol and Dave, are members of group1. Neither Carol nor Dave belong to any other group with a group stanza in the

LoadLeveler administration file, although they may use any string as the name of a LoadLeveler group and belong to it by default.

Because the **max_reservations** keyword is set to zero in the default group stanza, reservations can be made only in group1, which has an allotment of six reservations. Each reservation can have a maximum duration of 1440 minutes (24 hours).

Considering only the user-stanza attributes for reservations:
- User Carol can make up to four reservations with each having a maximum duration of 720 minutes (12 hours).
- User Dave can make up to four reservations with each having a maximum duration of 2880 minutes (48 hours).

If there are no reservations in the system and user Carol wants to make four reservations, she may do so. Each reservation can have a maximum duration of no more than 720 minutes. If Carol attempts to make a reservation with a duration greater than 720 minutes, LoadLeveler will not make the reservation because it exceeds the duration allowed for Carol.

Assume that Carol has created four reservations, and user Dave now wants to create four reservations:

- The number of reservations Dave may make is limited by the state of Carol's reservations and the maximum limit on reservations for group1. If the four reservations Carol made are still being set up, or are active, active shared or waiting, LoadLeveler will restrict Dave to making only two reservations at this time.
- Because the value of **max_reservation_duration** for the group is more restrictive than **max_reservation_duration** for user Dave, LoadLeveler enforces the group value, 1440 minutes.

If Dave belonged to another group that still had reservations available, then he could make reservations under that group, assuming the maximum number of reservations for the cluster had not been met. However, in this example, Dave cannot make any further reservations because they are allowed in group1 only.

Steps for integrating LoadLeveler with the Workload Manager

Another administrative setup task you must consider is whether you want to enforce resource usage of **ConsumableCpus**, **ConsumableMemory**, **ConsumableVirtualMemory**, and **ConsumableLargePageMemory**.

If you want to control these resources, Workload Manager (WLM) can be integrated with LoadLeveler to balance workloads at the machine level. When you are using WLM, workload balancing is done by assigning relative priorities to job processes. These job priorities prevent one job from monopolizing system resources when that resource is under contention.

To integrate LoadLeveler and WLM, perform the following steps:

1. As required for your use, define the applicable options for **ConsumableCpus**, **ConsumableMemory**, **ConsumableVirtualMemory**, or **ConsumableLargePageMemory** as consumable resources in the **SCHEDULE_BY_RESOURCES** global configuration keyword. This enables the LoadLeveler scheduler to consider these consumable resources.
2. As required for your use, define the applicable options for **ConsumableCpus**, **ConsumableMemory**, **ConsumableVirtualMemory**, or

ConsumableLargePageMemory in the **ENFORCE_RESOURCE_USAGE** global configuration keyword. This enables enforcement of these consumable resources by WLM.

3. Define **hard**, **soft** or **shares** in the **ENFORCE_RESOURCE_POLICY** configuration keyword. This defines what policy is used by LoadLeveler for CPUs and real memory when setting WLM class resource entitlements.
4. (Optional) Set the **ENFORCE_RESOURCE_MEMORY** configuration keyword to **true**. This setting allows WLM to limit the real memory usage of a WLM class as precisely as possible. When a class exceeds its limit, all processes in the class are killed.

   Rule: ConsumableMemory must be defined in the **ENFORCE_RESOURCE_USAGE** keyword in the global configuration file, or LoadLeveler does not consider the **ENFORCE_RESOURCE_MEMORY** keyword to be valid.

   Tips:

   - When set to true, the **ENFORCE_RESOURCE_MEMORY** keyword overrides the policy set through the **ENFORCE_RESOURCE_POLICY** keyword for **ConsumableMemory** only. The **ENFORCE_RESOURCE_POLICY** keyword value still applies for **ConsumableCpus**.
   - **ENFORCE_RESOURCE_MEMORY** may be set in either the global or the local configuration file. In the global configuration file, this keyword sets the default value for all the machines in the LoadLeveler cluster. If the keyword also is defined in a local file, the local setting overrides the global setting.
5. Using the **resources** keyword in a machine stanza in the administration file, define the CPU, real memory, virtual memory, and large page machine resources available for user jobs.
   - The **ConsumableCpus** reserved word accepts a count value of "all." This indicates that the initial resource count will be obtained from the Startd machine update value for CPUs.
   - If no resources are defined for a machine, then no enforcement will be done on that machine.
   - If the count specified by the administrator is greater than what the Startd update indicates, the initial count value will be reduced to match what the Startd reports.
   - For CPUs and real memory, if the count specified by the administrator is less than what the Startd update indicates, the WLM resource shares assigned to a job will be adjusted to represent that difference. In addition, a WLM softlimit will be defined for each WLM class. For example, if the administrator defines 8 CPUs on a 16 CPU machine, then a job requesting 4 CPUs will get a share of 4 and a softlimit of 50%.
   - Use caution when determining the amount of real memory available for user jobs. A certain percentage of a machine's real memory will be dedicated to the Default and System WLM classes and will not be included in the calculation of real memory available for users jobs.
     – On AIX, start LoadLeveler with the **ENFORCE_RESOURCE_USAGE** keyword enabled and issue **wlmstat -v -m**. Look at the npg column to determine how much memory is being used by these classes.
     – On Linux, start LoadLeveler with the **ENFORCE_RESOURCE_USAGE** keyword enabled and issue **cat /cgroup/memory/memory.usage_in_bytes** to determine how much memory is being used by these classes.
   - **ConsumableVirtualMemory** and **ConsumableLargePageMemory** are hard max limit values.
     – WLM considers the **ConsumableVirtualMemory** value to be real memory plus large page plus swap space.

– The **ConsumableLargePageMemory** value should be a value equal to the multiple of the pagesize. For example, 16MB (page size) * 4 pages = 64MB.

6. Decide if all jobs should have their CPU, real memory, virtual memory, or large page resources enforced and then define the **ENFORCE_RESOURCE_SUBMISSION** global configuration keyword.
   - If the value specified is **true**, LoadLeveler will check all jobs at submission time for the **resources** and **node_resources** keywords. To be submitted, either the job's **resources** or **node_resources** keyword must have the same resources specified as the **ENFORCE_RESOURCE_USAGE** keyword.
   - If the value specified is **false**, no checking is performed and jobs submitted without the **resources** or **node_resources** keyword will not have resources enforced and it might interfere with other jobs whose resources are enforced.
   - To support existing job command files without the **resources** or **node_resources** keyword, the **default_resources** and **default_node_resources** keywords in the class stanza can be defined.

For more information on the **ENFORCE_RESOURCE_USAGE** and the **ENFORCE_RESOURCE_SUBMISSION** keywords, see "Defining usage policies for consumable resources" on page 65.

LoadLeveler support for checkpointing jobs

Checkpointing is a method of periodically saving the state of a job step so that if the step does not complete it can be restarted from the saved state.

When checkpointing is enabled, checkpoints can be initiated from within the application at major milestones, or by the user, administrator or LoadLeveler external to the application.

Once a job step has been successfully checkpointed, if that step terminates before completion, the checkpoint file can be used to resume the job step from its saved state rather than from the beginning. When a job step terminates and is removed from the LoadLeveler job queue, it can be restarted from the checkpoint file by submitting a new job and setting the **restart_from_ckpt** = yes job command file keyword. When a job is terminated and remains on the LoadLeveler job queue, such as when a job step is vacated, the job step will automatically be restarted from the latest valid checkpoint file. A job can be vacated as a result of flushing a node, issuing checkpoint and hold, stopping or recycling LoadLeveler or as the result of a node crash.

To find out more about checkpointing jobs, use the information in Table 28.

Table 28. Roadmap of tasks for checkpointing jobs

Subtask	Associated instructions (see . . .)
Preparing the LoadLeveler environment for checkpointing and restarting jobs	• "Checkpoint keyword summary" on page 136 • "Planning considerations for checkpointing jobs" on page 136
Checkpointing and restarting jobs	• "Checkpointing a job" on page 226 • "Removing old checkpoint files" on page 144
Correctly specifying configuration and administration file keywords	• Chapter 10, "Configuration keyword reference," on page 231 • Chapter 11, "Administration keyword reference," on page 293

Checkpoint keyword summary

The following is a summary of keywords associated with the checkpoint and restart function.

- **Configuration file keywords**
  - **CKPT_CLEANUP_INTERVAL**
  - **CKPT_CLEANUP_PROGRAM**
  - **CKPT_EXECUTE_DIR**
  - **MAX_CKPT_INTERVAL**
  - **MIN_CKPT_INTERVAL**

  For more information about these keywords, see Chapter 10, "Configuration keyword reference," on page 231.
- **Administration file keywords**
  - **ckpt_dir**
  - **ckpt_time_limit**

  For more information about these keywords, see Chapter 11, "Administration keyword reference," on page 293.
- **Job command file keywords**
  - **checkpoint**
  - **ckpt_dir**
  - **ckpt_execute_dir**
  - **ckpt_subdir**
  - **ckpt_time_limit**
  - **restart_from_ckpt**

  For more information about these keywords, see "Job command file keyword descriptions" on page 335.

Planning considerations for checkpointing jobs

Review the following guidelines before you submit a checkpointing job:

- **Plan for jobs that you will restart on different nodes**

  If you plan to migrate jobs (restart jobs on a different node or set of nodes), you should understand the difference between writing checkpoint files to a local file system versus a global file system (such as AFS or GPFS™). The **ckpt_dir** and **ckpt_subdir** keywords in the job command file allow you to write to either type of file system. If you are using a local file system, before restarting the job from checkpoint, make certain that the checkpoint files are accessible from the machine on which the job will be restarted.

  POE provides the ability to checkpoint and later restart the entire set of programs that make up a parallel application. POE and LoadLeveler use the IBM MetaCluster Checkpoint Restart (MDCR) function, and its associated components, to coordinate the checkpointing and restarting of jobs. On AIX, MDCR invokes the appropriate Application Workload Partition (WPAR) commands on each node of a parallel job and coordinates the checkpoint and restart of those jobs. For more information about AIX WPAR, refer to the IBM AIX Information Center (**http://publib.boulder.ibm.com/infocenter/systems/scope/aix/index.jsp**).

  On AIX, MDCR and its associated components are installed with PE as part of the **installp** process. During the **installp** post-processing phase, PE adds entries to the **/etc/security/privcmds** file for the POE and PMD executables. For more information, see the topic about POE installation effects in *IBM Parallel Environment Runtime Edition: Installation*.

After POE installation, the system administrator needs to perform a number of tasks to provide users with the ability to checkpoint and restart parallel jobs. These tasks include the following:

– Authorizing users for checkpointing (AIX only)

– Defining the directories to be used for checkpointing

Notes:

1. Various limitations apply to checkpointing with PE. For example, checkpointing is only supported on User Space jobs. For a complete list of the restrictions, see *IBM Parallel Environment Runtime Edition: MPI Programming Guide*.
2. For more information about POE installation, see *IBM Parallel Environment Runtime Edition: Installation*.

- **Reserve adequate disk space**

  Checkpoint files require a significant amount of disk space. The checkpoint will fail if the directory where the checkpoint files are written does not have adequate space. Since the old set of checkpoint files are not deleted until the new set of files are successfully created, the checkpoint directory should be large enough to contain two sets of checkpoint files. You can make an accurate size estimate only after you have run your job and noticed the size of the checkpoint file that is created.

- **Plan for staging executables**

  If you want to stage the executable for a job step, use the **ckpt_execute_dir** keyword to define the directory where LoadLeveler will save the executable. This directory cannot be the same as the current location of the executable file, or LoadLeveler will not stage the executable.

  You may define the **ckpt_execute_dir** keyword in either the configuration file or the job command file. To decide where to define the keyword, use the information in Table 29.

Table 29. Deciding where to define the directory for staging executables

If the ckpt_execute_dir keyword is defined in:	Then the following information applies:
The configuration file only	• LoadLeveler stages the executable file in a new subdirectory of the specified directory. The name of the subdirectory is the job step ID. • The user is the owner of the subdirectory and has permission 700. • If the user issues the **llckpt** command with the **-k** option, LoadLeveler deletes the staged executable. • LoadLeveler will delete the subdirectory and the staged executable when the job step ends.
The job command file only	• LoadLeveler stages the executable file in the directory specified in the job command file. • The user is the owner of the file and has execute permission for it. • The user is responsible for deleting the staged file after the job step ends.
Both the configuration and job command files	
Neither file (the keyword is not defined)	LoadLeveler does not stage the executable file for the job step.

- **Set your checkpoint file size to the maximum**

To make sure that your job can write a large checkpoint file, assign your job to a job class that has its file size limit set to the maximum (unlimited). In the administration file, set up a class stanza for checkpointing jobs with the following entry:

```
file_limit = unlimited,unlimited
```

This statement specifies that there is no limit on the maximum size of a file that your program can create.

- **Choose a unique checkpoint file name**

  To prevent another job step from writing over your checkpoint file with another checkpoint file, make certain that your checkpoint file name is unique. The **ckpt_dir** and **ckpt_subdir** keywords give you control over the location and name of these files.

Additional planning considerations for checkpointing MetaCluster HPC jobs on AIX

Before you can checkpoint MetaCluster HPC jobs, you must install LoadLeveler for AIX and MetaCluster HPC.

For MetaCluster HPC installation information, see *Parallel Environment Runtime Edition for AIX: Installation* in the IBM Cluster Information Center (http://publib.boulder.ibm.com/infocenter/clresctr/vxrx/index.jsp).

Checkpoint and restart limitations

Use of the checkpoint and restart function has certain limitations. If you are planning to use the checkpoint and restart function, you need to be aware of the types of programs that cannot be checkpointed as well as the restrictions related to the operating system, nodes, tasks, threads, and so on.

For more information about checkpoint and restart limitations, see the *IBM Parallel Environment Runtime Edition MPI Programming Guide* in the IBM Cluster Information Center (http://publib.boulder.ibm.com/infocenter/clresctr/vxrx/index.jsp) or the IBM Publications Center (http://www.ibm.com/e-business/linkweb/publications/servlet/pbi.wss).

Submitting a MetaCluster HPC checkpoint job to LoadLeveler

Several LoadLeveler job command, configuration, and administration keywords are available to support the LoadLeveler/MetaCluster HPC checkpoint and restart operations.

They will be discussed in greater detail in other topics of this information. The purpose of this topic is to simply highlight the fact that a *regular* LoadLeveler job command file can be converted to a *checkpointable* job command file by adding the **checkpoint** and **ckpt_subdir** specifications to the file as shown in "job_1.cmd - A checkpointable job command file."

job_1.cmd - A checkpointable job command file

This checkpointable job command file may apply to your situation.

```
#!/bin/ksh
# # A parallel job command file using Metacluster HPC checkpoint/restart
# @ job_type=parallel
# @ step_name = ckpt_1
# @ class = small
```

```
# @ checkpoint = yes
# @ ckpt_subdir = /gpfs/cpr/job1
# @ initialdir = /home/llbld
# @ restart = no
# @ cpu_limit = 120
# @ input = /dev/null
# @ output = job1.$(Host).$(Cluster).$(Process).out
# @ error = job1.$(Host).$(Cluster).$(Process).err
# @ notification = error
# @ queue

/usr/bin/poe /gpfs/user/ckpt/ mpi_lapi_3 -t 4 -s 100 -ilevel 2 -pmdlog no
```

In the **job_1.cmd** file, the following has been added to inform LoadLeveler that the job is checkpointable:

```
# @ checkpoint = yes
```

To specify the directory where the checkpoint files will be stored, the following was also added:

```
# @ ckpt_subdir = /gpfs/cpr/job1
```

This job command file will be run as a shell script under LoadLeveler. It is, therefore, important that it has execute permission and that the first line of this file contains the string **#!/bin/ksh** or something similar (for example, **#!/bin/sh**) to indicate that it is to be run under the specified shell.

If the full path name of this file is **/gpfs./user/job_1.cmd**, you can use the following command to submit the job to LoadLeveler:

```
llsubmit /gpfs./user/job_1.cmd
```

You should receive a response similar to the following:

```
llsubmit: The job "c197blade2b11.ppd.pok.ibm.com.4" has been submitted.
```

If the **llq** command was used to query the LoadLeveler central manager for job status, the output of the **llq** command should be similar to the following:

```
Id                   Owner    Submitted   ST PRI Class     Running On
-------------------- -------- ----------- -- --- --------- -----------
c197blade2b11.2.0    llbld    11/2  14:42 R  50  small     c197blade2b11
c197blade2b11.3.0    llbld    11/2  14:42 R  50  small     c197blade2b12
c197blade2b11.4.0    llbld    11/2  14:42 R  50  small     c197blade2b12

3 job step(s) in queue, 0 waiting, 0 pending, 3 running, 0 held, 0 preempted
```

Note here an important difference in the way LoadLeveler handles checkpoint and noncheckpoint jobs. Because **job_1.cmd** is associated with a checkpoint job, LoadLeveler does not copy this script to the machine selected by LoadLeveler to run the job. This is in contrast to the case of a noncheckpoint job where the script is always copied to the LoadLeveler **execute** directory of the *running* machine. As a result, the **/gpfs./user/job_1.cmd** file must be accessible to all machines in the LoadLeveler cluster that can run this job. One way of accomplishing this objective is to put it on a shared file system. If this is not possible, you should consider using the **ckpt_execute_dir** keyword. For additional information on this keyword, see "Using the ckpt_execute_dir keyword" on page 144.

Using the llckpt command to checkpoint a job step

While the **c197blade2b11.4.0** job step is in the Running (**R**) state, you can issue a command to checkpoint this job step.

While the **c197blade2b11.4.0** job step is in the Running (**R**) state, you can issue the following command to checkpoint this job step:

```
llckpt c197blade2b11.4.0
```

You should receive output similar to the following:

```
llckpt: The llckpt command will wait for the results of the checkpoint operation \
        on job step c197blade2b11.4.0 before returning.
llckpt: Checkpoint of job step c197blade2b11.4.0 completed successfully.
```

Since the **llckpt c197blade2b11.4.0** command was run without any options, LoadLeveler will checkpoint the **c197blade2b11.4.0** job step, but this job step will continue to run after the checkpoint. The **llckpt** command has two major options that allow you to control what happens to a job step after a checkpoint:

- If the **-k** option is specified, the job step is terminated after a successful checkpoint.
- If the **-u** option is specified, the job step is put in user hold after a successful checkpoint.

Restarting a job step from a checkpoint

If a successful checkpoint of a job step has been made, you can request LoadLeveler to restart the job step from this checkpoint.

Restarting a job step from user hold

In this scenario, the command **llckpt -u c197blade2b11.4.0** has been executed to checkpoint the **c197blade2b11.4.0** job step and put it in user hold.

The output of the **llq** command should be similar to the following:

```
Id                   Owner    Submitted   ST PRI Class     Running On
-------------------- -------- ----------- -- --- --------- -----------
c197blade2b11.4.0    llbld    11/2  14:42 H  50  small

1 job step(s) in queue, 0 waiting, 0 pending, 0 running, 1 held, 0 preempted
```

To release **c197blade2b11.4.0** from user hold, issue the following command:

```
llhold -r c197blade2b11.4.0
```

LoadLeveler will dispatch the job step to a set of machines in the cluster that meets the job step requirements. These machine may or may not be the same machines that ran the job step before the checkpoint. On these machines, the job step resumes execution from the last successful checkpoint.

Running a new job to restart a job step from a checkpoint

If the **llckpt -k c197blade2b11.4.0** command has been used to checkpoint and terminate the **c197blade2b11.4.0** job step, you can resume execution of your application from this checkpoint by submitting a new job to LoadLeveler.

In the job command file of this new job, you need to specify the directory where the checkpoint files are located. To indicate that you want to run the new job from a checkpoint, this job command file must also contain the following statement:

```
# @ restart_from_ckpt = yes
```

"restart_job_1.cmd - A job command file to restart a job step from a checkpoint" shows the content of such a file (**restart_job_1.cmd**).

restart_job_1.cmd - A job command file to restart a job step from a checkpoint:

This job command file to restart a job step from a checkpoint may apply to your situation.

```
#!/bin/ksh
# # A parallel job command file using Metacluster HPC checkpoint/restart
# @ job_type=parallel
# @ step_name = ckpt_1
# @ class = small
# @ checkpoint = yes
# @ restart_from_ckpt = yes
# @ ckpt_subdir = /gpfs/cpr/job1
# @ initialdir = /home/llbld
# @ restart = no
# @ cpu_limit = 120
# @ input = /dev/null
# @ output = job1.$(Host).$(Cluster).$(Process).out
# @ error = job1.$(Host).$(Cluster).$(Process).err
# @ notification = error
# @ queue

/usr/bin/poe /gpfs/user/ckpt/ mpi_lapi_3 -t 4 -s 100 -ilevel 2 -pmdlog no
```

Notes:

1. This file is identical to the **job_1.cmd** file except for the addition of the statement **# @ restart_from_ckpt = yes**.
2. The **ckpt_subdir** keyword specifies the directory where the checkpoint files are located.
3. The checkpoint keyword is set to **yes**. This means that this new job can be checkpointed again if needed.

Making a duplicate of the original job command file and then adding the statement **# @ restart_from_ckpt = yes** to the copy is a fast and a safe way to prepare a new job for restart operations. However, it should be noted that since almost all of the information needed to restart a job step is contained in the files in the checkpoint directory, the **restart_job_1.cmd** file in this example can be simplified. “restart_job_1B.cmd - A simplified job command file to restart a job step from a checkpoint” shows the content of a simplified restart file.

restart_job_1B.cmd - A simplified job command file to restart a job step from a checkpoint:

This simplified job command file to restart a job step from a checkpoint may apply to your situation.

```
#!/bin/ksh
# # A parallel job command file using Metacluster HPC checkpoint/restart
# @ job_type=parallel
# @ step_name = ckpt_1
# @ class = small
# @ checkpoint = yes
# @ restart_from_ckpt = yes
# @ ckpt_subdir = /gpfs/cpr/job1
# @ initialdir = /home/llbld
# @ restart = no
# @ cpu_limit = 120
# @ input = /dev/null
# @ output = job1.$(Host).$(Cluster).$(Process).out
# @ error = job1.$(Host).$(Cluster).$(Process).err
# @ notification = error
# @ queue
```

In this example, you can run either of the following commands to restart from the checkpoint file:

```
llsubmit restart_job_1.cmd
```

or

```
llsubmit restart_job_1B.cmd
```

However, in both cases, it is important that the **/gpfs./user/job_1.cmd** file that was run before the checkpoint is still in its original location and is unmodified at restart time. This is because on restart, the script that is actually run is **job_1.cmd** not **restart_job_1.cmd** or **restart_job_1B.cmd**.

Making periodic checkpoints

LoadLeveler supports periodic checkpointing of user applications if the configuration file **MIN_CKPT_INTERVAL** and **MAX_CKPT_INTERVAL** keywords are defined and the job command file **checkpoint** keyword is set to the value **interval**.

Example - checkpoint every 30 seconds

Using this example, when the job is run under LoadLeveler, a checkpoint will be made every 30 seconds.

If the LoadLeveler global configuration file contains the following statements:

```
MIN_CKPT_INTERVAL    = 30
MAX_CKPT_INTERVAL    = 30
```

and the job command file contains the statements:

```
# @ checkpoint = interval
# @ ckpt_subdir = /gpfs/cpr/job1
```

then when the job is run under LoadLeveler, a checkpoint will be made every 30 seconds. At each checkpoint, the checkpoint information of the job is saved at one of the following directories:

```
/gpfs/cpr/job1/ckpt_0
/gpfs/cpr/job1/ckpt_1
/gpfs/cpr/job1/ckpt_current -> /gpfs/cpr/job1/ckpt_0
```

The two directories are reused in an endless loop. The **ckpt_current** file is a symbolic link pointing to the directory containing the last successful checkpoint.

Example - checkpoint interval

This example of the checkpoint interval may apply to your situation.

If the LoadLeveler global configuration file contains the following statements:

```
MIN_CKPT_INTERVAL    = 30
MAX_CKPT_INTERVAL    = 300
```

and the job command file contains the statement:

```
# @ checkpoint = interval
# @ ckpt_subdir = /gpfs/cpr/job1
```

then when the job is run under LoadLeveler, a first checkpoint will be made after 30 seconds. The second checkpoint is made after a time **interval = 2 x MIN_CKPT_INTERVAL**. The time interval between checkpoints will keep on doubling in value until the **MAX_CKPT_INTERVAL** value of 300 seconds is

reached. As in "Example - checkpoint every 30 seconds" on page 142, the checkpoint information of the job is saved at either **/gpfs/cpr/job1/ckpt_0** or **ckpt_1** with the **ckpt_current** file pointing to the last successful checkpoint.

Using the ckpt_dir and ckpt_subdir keywords

The **ckpt_dir** keyword is both a LoadLeveler administration file keyword and a job command file keyword. The **ckpt_subdir** keyword is a job command file keyword only.

The full path name of the directory used to store the LoadLeveler/MetaCluster checkpoint information is a concatenation of the values of **ckpt_dir** and **ckpt_subdir**. In this topic, the use of these keywords is illustrated with a number of examples.

Example - storing checkpoint information for the job step in the /gpfs/user_1/ckpt_test1 directory

This example of storing checkpoint information for the job step in the /gpfs/user_1/ckpt_test1 directory may apply to your situation.

In this example, the job command file contains the following specifications:

```
# @ ckpt_dir  = /gpfs/MetaC/CKPT
# @ ckpt_subdir = /gpfs/user_1/ckpt_test1
```

Note that the value of **ckpt_subdir** is a string starting with "**/**". Since **ckpt_subdir** specifies a fully qualified path name, the **ckpt_dir** keyword is ignored. LoadLeveler stores checkpoint information for the job step in the **/gpfs/user_1/ckpt_test1** directory.

Example - storing checkpoint information for the job step in the /gpfs/MetaC/CKPT/ckpt_test1 directory

This example of storing checkpoint information for the job step in the /gpfs/MetaC/CKPT/ckpt_test1 directory may apply to your situation.

In this example, the job command file contains the following specifications:

```
# @ ckpt_dir  = /gpfs/MetaC/CKPT
# @ ckpt_subdir = ckpt_test1
```

LoadLeveler stores checkpoint information for the job step in the **/gpfs/MetaC/CKPT/ckpt_test1** directory, which is a concatenation of the values associated with **ckpt_dir** and **ckpt_subdir**.

Example - storing checkpoint information for the job in the /gpfs/MetaC/CKPT_small/ckpt_test1 directory

This example of storing checkpoint information for the job in the /gpfs/MetaC/CKPT_small/ckpt_test1 directory may apply to your situation.

In this example, the class stanza of the class **small** in the LoadLeveler administration file contains this specification:

```
ckpt_dir = /gpfs/MetaC/CKPT_small
```

The job command file contains the following specifications:

```
# @ ckpt_subdir = ckpt_test1
# @ class = small
```

LoadLeveler stores checkpoint information for the job in the **/gpfs/MetaC/CKPT_small/ckpt_test1** directory, which is a concatenation of the value associated with **ckpt_dir** for class **small** and the value of **ckpt_subdir**.

Example - storing checkpoint information for the job in the /gpfs/MetaC/test99.hostname3.pok.ibm.com.905.1.ckpt directory

In this example, the **ckpt_dir** keyword is not specified in the LoadLeveler administration file or the job command file.

The **ckpt_subdir** keyword is not specified in the job command file. The job command file contains the following specifications:

```
# @ job_name = test99
# @ class = small_job
# @ checkpoint = yes
# @ initialdir = /gpfs/MetaC
```

Assuming that the job is run with the LoadLeveler assigned job step ID of **hostname3.pok.ibm.com.905.1**, LoadLeveler stores checkpoint information for the job in the **/gpfs/MetaC/test99.hostname3.pok.ibm.com.905.1.ckpt** directory. This is because the default value of **ckpt_dir** is the initial working directory and the default value of **ckpt_subfile** is **[***jobname***].***job_step_id***.ckpt**.

Note: You must specify a value for **ckpt_subdir** when **restart_from_ckpt=yes**. LoadLeveler cannot generate a default value for **ckpt_subdir** on restart because any default would be based on the job step ID of this new job and **ckpt_subdir** for a restarted job must point to the location of the checkpoint files of some previously run job.

Removing old checkpoint files

To keep your system free of checkpoint files that are no longer necessary, LoadLeveler provides two keywords to help automate the process of removing these files:

- **CKPT_CLEANUP_PROGRAM**
- **CKPT_CLEANUP_INTERVAL**

Both keywords must contain valid values to automate this process. For information about configuration file keyword syntax and other details, see Chapter 10, “Configuration keyword reference,” on page 231.

Using the ckpt_execute_dir keyword

The **ckpt_execute_dir** keyword is both a LoadLeveler configuration file keyword and a job command file keyword.

When used as a job command file keyword, it specifies the directory where the job step executable will be saved for a checkpointable job. In this topic, the use of this keyword as a job command file keyword is illustrated with a number of examples.

Example - using the llsubmit job_2.cmd

In this file, both the **executable** keyword and the **ckpt_execute_dir** keywords are specified.

“job_2.cmd - A parallel checkpoint job using the executable keyword and the ckpt_execute_dir keyword” on page 145 shows the content of the **job_2.cmd** file. When this job is submitted to LoadLeveler with the command:

```
llsubmit job_2.cmd
```

the **/home/llbld/my_bin/my_application** file is copied at dispatch time to the **/gpfs/user/ckpt_bin** directory and **/gpfs/user/ckpt_bin/my_application** is the full path name of the application that will be run by LoadLeveler.

If a checkpoint and terminate operation is made with the **llckpt -k** *job_step_id* command, the checkpoint information is saved in the **/gpfs/cpr/job2** directory. The **/gpfs/user/ckpt_bin/my_application** file is *not* deleted by LoadLeveler because it may be needed for restart operations. It is your responsibility to manage the files in the **ckpt_execute_dir** directory and to remove any files that are no longer needed.

job_2.cmd - A parallel checkpoint job using the executable keyword and the ckpt_execute_dir keyword:

This parallel checkpoint job using the executable keyword and the ckpt_execute_dir keyword may apply to your situation.

```
#!/bin/ksh
# @ job_type=parallel
# @ step_name = ckpt_1
# @ class = small
# @ checkpoint = yes
# @ ckpt_subdir = /gpfs/cpr/job2
# @ executable = /home/llbld/my_bin/my_application
# @ ckpt_execute_dir = /gpfs/user/ckpt_bin
# @ initialdir = /home/llbld
# @ restart = no
# @ cpu_limit = 120
# @ input  = /dev/null
# @ output = job1.$(Host).$(Cluster).$(Process).out
# @ error  = job1.$(Host).$(Cluster).$(Process).err
# @ notification = error
# @ queue
```

Example - using the llsubmit job_3.cmd

Since the **executable** keyword is not specified, the **job_3.cmd** script itself is the *executable*.

"job_3.cmd - A parallel checkpoint job using the ckpt_execute_dir keyword, but not the executable keyword" shows the content of the **job_3.cmd** file.

In this example, **ckpt_execute_dir** has the value **/gpfs/user/ckpt_bin**. When this job is submitted to LoadLeveler with the command:

```
llsubmit job_3.cmd
```

the **job_3.cmd** file is copied to the **/gpfs/user/ckpt_bin** directory at dispatch time and value **/gpfs/user/ckpt_bin /job_3.cmd** is the full path name of the application that will be run by LoadLeveler.

As in "Example - using the llsubmit job_2.cmd" on page 144, when this job terminates the **/gpfs/user/ckpt_bin /job_3.cmd** file is not deleted by LoadLeveler because it may be needed for restart operations. It is your responsibility to manage the files in the **ckpt_execute_dir** directory and to remove any files that are no longer needed.

job_3.cmd - A parallel checkpoint job using the ckpt_execute_dir keyword, but not the executable keyword:

This parallel checkpoint job using the ckpt_execute_dir keyword, but not the executable keyword may apply to your situation.

```
#!/bin/ksh
# @ job_type=parallel
# @ step_name = ckpt_1
# @ class = small
# @ checkpoint = yes
# @ ckpt_subdir = /gpfs/cpr/job2
# @ ckpt_execute_dir = /gpfs/user/ckpt_bin
# @ initialdir = /home/llbld
# @ restart = no
# @ cpu_limit = 120
# @ input = /dev/null
# @ output = job1.$(Host).$(Cluster).$(Process).out
# @ error = job1.$(Host).$(Cluster).$(Process).err
# @ notification = error
# @ queue

/usr/bin/poe /gpfs/user/ckpt/ mpi_lapi_3 -t 4 -s 100 -ilevel 2 -pmdlog no
```

Initiating a checkpoint using the ll_ckpt() API

You can initiate a checkpoint by running the command **llckpt** *job_step_id*.

As an alternative, you can use the LoadLeveler **ll_ckpt()** interface. This interface allows you to perform almost all the functions of the **llckpt** command from within your C or C++ application. The **test_ll_ckpt.c** program shows how the **ll_ckpt()** interface can be used to initiate a checkpoint from an application written in C.

The test_ll_ckpt.c program

The test_ll_ckpt.c program may apply to your situation.

```
/* ll_test_ckpt.c: A C program showing how the LoadLeveler ll_ckpt() interface */
/* can be used to initiate the checkpoint of a LoadLeveler job step.           */
/* Compile and Link:                                                           */
/*   g++ -c -I/opt/ibmll/LoadL/full/include -DMETACLUSTER_CKPT  test_ll_ckpt.c */
/*   g++ -o test_ll_ckpt test_ll_ckpt.o -lllapi -lpthread -ldl                 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <signal.h>
#include "llapi.h"

main(int argc, char *argv[])
{
   LL_ckpt_info   *ckpt_info = NULL;
   char *job_step_id;
   int rc;

   if (argc != 2) {
      printf("Syntax: job_step_id>\n", argv[0]);
      exit(-1);
   }
   job_step_id = strdup(argv[1]);

   if((ckpt_info = (LL_ckpt_info*)malloc(sizeof(LL_ckpt_info))) == NULL) }
      printf("Unable to allocate memory.\n");
      exit(-1);
   }
   if((ckpt_info->cp_error_data =
         (cr_error_t*)malloc(sizeof(cr_error_t))) == NULL) {
      printf("Unable to allocate memory.\n");
      exit(-1);
   }
```

```
   /* Initialize the info_ckpt structure to some default values. */
   ckpt_info->version = LL_API_VERSION;
   ckpt_info->step_id = NULL;
   ckpt_info->waitType = CKPT_WAIT;
   ckpt_info->ckptType = CKPT_AND_CONTINUE;
   ckpt_info->abort_sig = SIGINT;
   ckpt_info->cp_error_data->Py_error = 0;
   ckpt_info->cp_error_data->Sy_error = 0;
   ckpt_info->cp_error_data->Xtnd_error = 0;
   ckpt_info->cp_error_data->error_len = 0;
   ckpt_info->cp_error_data->error_data = NULL;
   ckpt_info->soft_limit = 0;
   ckpt_info->hard_limit = 0;

   /* We will checkpoint and terminate the job step specified. */
   ckpt_info->step_id = job_step_id;
   ckpt_info->ckptType = CKPT_AND_TERMINATE;
   rc = ll_ckpt(ckpt_info);
   if (rc == 0) {
      printf("Checkpoint operation is successful.\n");
   }
   else {
      printf("Checkpoint operation failed.\n");
      if (ckpt_info->cp_error_data->error_data) {
         printf("Error message: cp_error_data->error_data);
      }
   }
   return (0);
}
```

LoadLeveler scheduling affinity support

LoadLeveler offers a number of scheduling affinity options.

The LoadLeveler scheduling affinity options are:
- Memory affinity
- Adapter affinity
- Processor affinity

Enabling scheduling affinity allows LoadLeveler jobs to utilize performance improvement from memory, adapter, and processor affinities. On IBM POWER7® and POWER6® processor-based systems, memory and adapter affinity are provided using multiple chip modules (MCMs). On Linux x86 platforms, memory affinity is provided using the Non-Uniform Memory Access (NUMA) architecture. If enabled, LoadLeveler will schedule and attach the appropriate CPUs in the cluster to the job tasks in order to maximize performance improvement based on the type of affinity requested by the job.

Memory affinity

IBM POWER7 and POWER6 processor-based systems contain MCMs, each containing multiple processors. System memory is attached to these MCMs. While any processor can access all of the memory in the system, a processor has faster access and higher bandwidth when addressing memory that is attached to its own MCM rather than memory attached to the other MCMs in the system. For more information about memory affinity, see the *AIX Performance Management Guide*.

On Linux x86 platforms, with the NUMA architecture, a processor can access its own local memory faster than non-local memory (memory local to another processor or memory shared between processors).

Adapter affinity

On IBM POWER processor-based systems, the concept of affinity also applies to the I/O subsystem. The processes running on CPUs from an MCM have faster access to the adapters attached to the I/O slots of that MCM. I/O affinity will be referred to as adapter affinity in this topic. For more information about adapter affinity, see the *AIX Performance Management Guide*.

Processor affinity

LoadLeveler provides processor affinity options to improve job performance on the following platforms:

- IBM POWER7 processor-based systems running in simultaneous multithreading (SMT) mode with AIX or Linux
- IBM POWER7 processor-based systems running in Single Threaded (ST) mode with AIX or Linux
- x86 and x86_64 processor-based systems with Linux

On AIX, affinity support is implemented by using a Resource Set (RSet), which contains bit maps for CPU and memory pool resources. The RSet APIs available in AIX can be used to attach RSets to processes. Attaching an RSet to a process limits the process to only using the resources contained in the RSet. One of the main uses of RSets is to limit the application processes to run only on the processors contained in a single MCM and hence to benefit from memory affinity. For more details on RSets, refer to *AIX System Management Guide: Operating System and Devices*.

On Linux systems, affinity support is implemented by using "cpusets," which provide a mechanism for assigning a set of CPUs and memory nodes (MCMs) to a set of tasks. The cpusets constrain the CPU and memory placement of tasks to only the resources within a task's current cpuset. The cpusets are managed by the virtual file system type cpuset. Before configuring LoadLeveler to support affinity, the cpuset virtual file system must be created on every machine in the cluster to enable affinity support.

Configuring LoadLeveler to use scheduling affinity

On AIX and Linux systems, scheduling affinity can be enabled by using the **RSET_SUPPORT** configuration file keyword. Machines that are configured with this keyword indicate the ability to service jobs requesting or requiring scheduling affinity.

Enable **RSET_SUPPORT** with one of these values:

- Choose **RSET_MCM_AFFINITY** to allow jobs specifying **rset = RSET_MCM_AFFINITY** or the **task_affinity** keyword to run on a node. When **rset = RSET_MCM_AFFINITY**, LoadLeveler will select and attach sets of CPUs to task processes such that a set of CPUs will be from the same MCM. When the **task_affinity** keyword is used, LoadLeveler will select CPUs regardless of their location with respect to an MCM.
- Choose **RSET_USER_DEFINED** to allow jobs specifying a user-defined RSet name for **rset** to run on a node. The **RSET_USER_DEFINED** option enables scheduling affinity, allowing users more control over scheduling affinity parameters by allowing the use of user-defined RSets. Through the use of user-defined RSets, users can utilize new RSet features before a LoadLeveler

implementation is released. This option also allows users to specify a different number of CPUs in their RSets depending on the needs of each task. This value is supported only on AIX machines.

Notes:

1. Because LoadLeveler creates a cpuset for each task requesting affinity under the **/dev/cpuset** directory on Linux machines, the cpuset virtual file system must be created and mounted on the **/dev/cpuset** directory by issuing the following commands on each node:

   ```
   # mkdir /dev/cpuset
   # mount -t cpuset none /dev/cpuset
   ```

2. A virtual file system of type cpuset mounted at **/dev/cpuset** will be deleted when the node is rebooted. To create the **/dev/cpuset** directory and have the virtual cpuset file system mounted on it automatically when the node is rebooted, add the following commands to your start-up script (for example, **/etc/init.d/boot.local**), which is run when the node is rebooted or started:

   ```
   if test -e /dev/cpuset || mkdir -p /dev/cpuset ; then
    mount -t cpuset none /dev/cpuset
   fi
   ```

See "Configuration keyword descriptions" on page 233 for more information on the **RSET_SUPPORT** keyword.

On AIX and Linux systems, jobs requesting processor affinity with the **task_affinity** keyword in the job command file will only run on machines where the resource statement in the machine stanza in the LoadLeveler administration file contains the **ConsumableCpus** keyword. For more information on specifying **ConsumableCpus**, see the **resource** keyword description in "Administration keyword descriptions" on page 298.

LoadLeveler multicluster support

To provide a more scalable runtime environment and more efficient workload balancing, you may configure a LoadLeveler multicluster environment.

A LoadLeveler multicluster environment consists of two or more LoadLeveler clusters, grouped together through network connections that allow the clusters to share resources. These clusters may be AIX, Linux, or mixed clusters.

Within a LoadLeveler multicluster environment:

- The **local cluster** is the cluster from which the user submits jobs or issues commands.
- A **remote cluster** is a cluster that accepts job submissions and commands from the local cluster.
- A **local gateway Schedd** is a Schedd within the local cluster serving as an inbound point from some remote cluster, an outbound point to some remote cluster, or both.
- A **remote gateway Schedd** is a Schedd within a remote cluster serving as an inbound point from the local cluster, an outbound point to the local cluster, or both.
- A **local central manager** is the central manager in the same cluster as the local gateway Schedd.
- A **remote central manager** is the central manager in the same cluster as a remote gateway Schedd.

A LoadLeveler multicluster environment addresses scalability and workload balancing issues by providing the ability to:

- Distribute workload among LoadLeveler clusters when jobs are submitted.
- Easily access multiple LoadLeveler cluster resources.
- Display information about the multicluster.
- Monitor and control operations in a multicluster.
- Transfer idle jobs from one cluster to another.
- Transfer user input and output files between clusters.
- Enable LoadLeveler to operate in a secure environment where clusters are separated by a firewall.

Table 30 shows the multicluster support subtasks with a pointer to the associated instructions:

Table 30. Multicluster support subtasks and associated instructions

Subtask	Associated instructions (see . . .)
Configure a LoadLeveler multicluster	"Configuring a LoadLeveler multicluster"
Submit and monitor jobs in a LoadLeveler multicluster	"Submitting and monitoring jobs in a LoadLeveler multicluster" on page 218

Table 31 shows multicluster support related topics:

Table 31. Multicluster support related topics

Related topics	Additional information (see . . .)
Administration file: Cluster stanzas	"Defining clusters" on page 104
Administration file: Cluster keywords	"Administration keyword descriptions" on page 298
Configuration file: Cluster keywords	"Configuration keyword descriptions" on page 233
Job command file: Cluster keywords	"Job command file keyword descriptions" on page 335
Commands and APIs	*LoadLeveler: Command and API Reference*
Diagnosis and messages	*LoadLeveler: Diagnosis and Messages Guide*

Configuring a LoadLeveler multicluster

These are the subtasks for configuring a LoadLeveler multicluster.

Table 32 lists the subtasks for configuring a LoadLeveler multicluster.

Table 32. Subtasks for configuring a LoadLeveler multicluster

Subtask	Associated instructions (see . . .)
Configure the LoadLeveler multicluster environment	• "Steps for configuring a LoadLeveler multicluster" on page 151 • "Steps for securing communications within a LoadLeveler multicluster" on page 153

Table 32. Subtasks for configuring a LoadLeveler multicluster (continued)

Subtask	Associated instructions (see . . .)
Display information about the LoadLeveler multicluster environment	• Use the **llstatus** command: – With the **-X** option to display information about machines in the multicluster. – With the **-C** option to display information defined in cluster stanzas in the administration file. • Use the **llclass** command with the **-X** option to display information about classes on any cluster (local or remote). • Use the **llq** command with the **-X** option to display information about jobs on any cluster (local or remote).
Monitor and control operations in the LoadLeveler multicluster environment	Existing LoadLeveler user commands accept the **-X** option for a multicluster environment. **Rules:** • Administrator only commands are not applicable in a multicluster environment. • The options **-x**, **-W**, **-s**, and **-p** cannot be specified together with the **-X** option on the **llmodify** command. • The options **-x** and **-w** cannot be specified together with the **-X** option on the **llq** command. • The **-X** option on the following commands is restricted to a single cluster: – **llcancel** – **llckpt** – **llhold** – **llmodify** – **llprio** • The following commands are not applicable in a multicluster environment: – **llacctmrg** – **llchres** – **llinit** – **llmkres** – **llqres** – **llrmres** – **llrunscheduler** – **llsummary**

Steps for configuring a LoadLeveler multicluster

The primary task for configuring a LoadLeveler multicluster environment is to enable communication between gateway Schedd daemons on all of the clusters in the multicluster. To do so requires defining each Schedd daemon as either local or remote, and defining the inbound and outbound hosts with which the daemon will communicate.

Before you begin: You need to know that:
- A single machine may be defined as an inbound or outbound host, or as both.
- A single cluster must belong to only one multicluster.
- A single multicluster must consist of 10 or fewer clusters.
- Clusters must have unique host names within the multicluster network domain space.
- The inbound Schedd becomes the **schedd_host** of all remote jobs it receives.

Perform the following steps to configure a LoadLeveler multicluster:

1. In the administration file, define one cluster stanza for each cluster in the LoadLeveler multicluster environment.

   Rules:
   - You must define one cluster as the local cluster.
   - You must code the following required cluster-stanza keywords and variable values:

     cluster_name: **type=cluster**
     outbound_hosts = *hostname*[(*cluster_name*)]
     inbound_hosts = *hostname*[(*cluster_name*)]
   - If you want to allow users to submit remote jobs to the local cluster, the list of inbound hosts must include the name of the inbound Schedd and the cluster you are defining as remote or you must specify the name of an inbound Schedd without any cluster specification so that it defaults to being an inbound Schedd for all clusters.
   - If the configuration file keyword **SCHEDD_STREAM_PORT** for any cluster is set to use a port other than the default value of 9605, you must set the **inbound_schedd_port** keyword in the cluster stanza for that cluster.
2. (Optional) If the local cluster wants to provide job distribution where users allow LoadLeveler to select the appropriate cluster for job submission based on administration defined objectives, then define an installation exit to be executed at submit time using the **CLUSTER_METRIC** configuration keyword. You can use the LoadLeveler data access APIs in this exit to query other clusters for information about possible metrics, such as the number of jobs in a specified job class, the number of jobs in the idle queue, or the number of free nodes in the cluster. For more information, see "Configuration keyword descriptions" on page 233.

   Tip: LoadLeveler provides a set of sample exits for you to use as models. These samples are in the `${RELEASEDIR}/samples/llcluster` directory.
3. (Optional) If the local cluster wants to perform user mapping on jobs arriving from remote clusters, define the **CLUSTER_USER_MAPPER** configuration keyword. For more information, see "Configuration keyword descriptions" on page 233.
4. (Optional) If the local cluster wants to perform job filtering on jobs received from remote clusters, define the **CLUSTER_REMOTE_JOB_FILTER** configuration keyword. For more information, see "Configuration keyword descriptions" on page 233.
5. Notify LoadLeveler daemons by issuing the **llctl** command with either the **reconfig** or **recycle** keyword. Otherwise, LoadLeveler will not process the modifications you made to the administration file.

Additional considerations:

- Remote jobs are subjected to the same configuration checks as locally submitted jobs. Examples include account validation, class limits, include lists, and exclude lists.
- Remote jobs will be processed by the local **submit_filter** prior to submission to a remote cluster.
- Any tracker program specified in the API parameters will be invoked upon the scheduling cluster nodes.
- If a step is enabled for checkpoint and the **ckpt_execute_dir** is not specified, LoadLeveler will not copy the executable to the remote cluster, the user must ensure that executable exists on the remote cluster. If the executable is not in a shared file system, the executable can be copied to the remote cluster using the **cluster_input_file** job command file keyword.

- If the job command file is also the executable and the job is submitted or moved to a remote cluster, the **$(executable)** variable will contain the full path name of the executable on the local cluster from which it came. This differs from the behavior on the local cluster, where the **$(executable)** variable will be the command line argument passed to the **llsubmit** command. If you only want the file name, use the **$(base_executable)** variable.

Steps for securing communications within a LoadLeveler multicluster

Configuring LoadLeveler to use the OpenSSL library enables it to operate in a secure environment where clusters are separated by a firewall.

Perform the following steps to configure LoadLeveler to use OpenSSL in a multicluster environment:

1. Install SSL using the standard platform installation process.
2. Ensure a link exists from the installed SSL library to:
   a. **/usr/lib/libssl.so** for 32-bit Linux platforms.
   b. **/usr/lib64/libssl.so** for 64-bit Linux platforms.
   c. **/usr/lib/libssl.a** for AIX platforms.
3. Create the SSL authorization keys by invoking the **llclusterauth** command with the **-k** option on all local gateway schedds.

   Result: LoadLeveler creates a public key, a private key, and a security certificate for each gateway node.
4. Distribute the public keys to remote gateway schedds on other secure clusters. This is done by exchanging the public keys with the other clusters you wish to communicate with.
   - for AIX, public keys can be found in the **/var/LoadL/ssl/id_rsa.pub** file.
   - for Linux, public keys can be found in the **/var/opt/LoadL/ssl/id_rsa.pub** file.
5. Copy the public keys of the clusters you wish to communicate with into the **authorized_keys** directory on your inbound Schedd nodes.
   - for AIX, **/var/LoadL/ssl/authorized_keys**
   - for Linux, **/var/opt/LoadL/ssl/authorized_keys**
   - The authorization key files can be named anything within the **authorized_keys** directory.
6. Define the cluster stanzas within the LoadLeveler administration file, using the **multicluster_security = SSL** keyword. Define the keyword **ssl_cipher_list** if a specific OpenSSL cipher encryption method is desired. Use **secure_schedd_port** to define the port number to be used for secure inbound transactions to the cluster.
7. Notify LoadLeveler daemons by issuing the **llctl -g** command with the **recycle** keyword. Otherwise, LoadLeveler will not process the modifications you made to the administration file.
8. Configure firewalls to accept connections to the **secure_schedd_port** numbers you defined in the administration file.

LoadLeveler Blue Gene support

Blue Gene is a massively parallel system based on a scalable cellular architecture which exploits a very large number of tightly interconnected compute nodes (C-nodes).

To take advantage of Blue Gene support, you must be using the LoadLeveler BACKFILL scheduler. With the BACKFILL scheduler, LoadLeveler enables the Blue Gene system to take advantage of reservations that allow you to schedule when, and with which resources a job will run.

Terms you should know:

Front End Nodes (FEN)
Machines from which users and administrators interact with Blue Gene. Applications are compiled on and submitted for execution in the Blue Gene compute nodes from FENs. User interactions with applications, including debugging, are also performed from the FENs.

Service nodes
Dedicated hardware that runs software to control and manage the Blue Gene system.

I/O nodes
Special nodes that connect the compute nodes to the outside world. I/O nodes allow processes that are executing in the compute nodes to perform I/O operations, such as accessing files, and to communicate with the job management system.

Control System
A component that serves as the interface to the Blue Gene system. It contains persistent storage with configuration and status information on the entire system. It also provides various services to perform actions on the Blue Gene system, such as launching a job.

Compute nodes (also called C-nodes)
The primary computational resource for execution of user applications. Compute nodes run a custom lightweight kernel called CNK.

Node boards
An intermediate packaging component of Blue Gene consisting of 32 compute nodes.

Midplanes
The basic scalable unit of a Blue Gene system consisting of 16 node boards making up 512 compute nodes.

block A group of compute nodes allocated to execute a job. Blocks are physically (electronically) isolated from each other (for example, messages cannot flow outside an allocated block) and can have connectivity of Mesh or Torus.

small block
A group of compute nodes requested that is smaller than one midplane. Valid small block sizes are 32, 64, 128, and 256 compute nodes.

runjob
The interface to launch jobs on Blue Gene/Q, replacing **mpirun**.

For more information about the Blue Gene system and Blue Gene terminology, refer to IBM System Blue Gene Solution documentation on IBM Redbooks® (http://www.redbooks.ibm.com/).

Table 33 on page 155 lists the Blue Gene subtasks with a pointer to the associated instructions:

Table 33. Blue Gene subtasks and associated instructions

Subtask	Associated instructions (see . . .)
Configure LoadLeveler Blue Gene support	"Configuring LoadLeveler Blue Gene support"
Submit and monitor Blue Gene jobs	See *LoadLeveler: Using and Administering*

Table 34 lists the Blue Gene related topics and associated information:

Table 34. Blue Gene related topics and associated information

Related topic	Associated information (see . . .)
Configuration file: Blue Gene keywords	Chapter 10, "Configuration keyword reference," on page 231
Job command file: Blue Gene keywords	See *LoadLeveler: Using and Administering*
Commands and APIs	*LoadLeveler: Command and API Reference*
Diagnosis and messages	*LoadLeveler: Diagnosis and Messages Guide*

Configuring LoadLeveler Blue Gene support

Table 35 lists the subtasks for configuring LoadLeveler Blue Gene support along with a pointer to the associated instructions:

Table 35. Blue Gene configuring subtasks and associated instructions

Subtask	Associated instructions (see . . .)
Configuring LoadLeveler Blue Gene support	"Steps for configuring LoadLeveler Blue Gene support"
Display information about the Blue Gene system	• Use the **llbgstatus** command for information about the Blue Gene system. The **-B** or **-M** options can be used to display information about Blue Gene blocks or midplanes. See *LoadLeveler: Command and API Reference* for information about the **llbgstatus** command.
Display information about Blue gene jobs	• Use the **llsummary** command with the **-l** option to display job resource information. • Use the **llq** command with the **-b** option to display information about all Blue Gene jobs.

Steps for configuring LoadLeveler Blue Gene support

In order for LoadLeveler to operate correctly in the Blue Gene/Q environment, the LoadLeveler plugin must be specified as the plugin for IBM runjob mux. This can be done by adding the plugin under the **runjob.mux** section in the **bg.properties** file:

```
[runjob.mux]
...
plugin = /usr/lib64/libllrunjob_mux.so
    # Fully qualified path to the plugin used for communicating with a job scheduler.
    # This value can be updated by the runjob_mux_refresh_config command on the
    # Login Node where a runjob_mux process runs.
...
```

When updates are applied for the LoadLeveler software, you should refresh the **runjob.mux** by using the **runjob_mux_refresh_config** command.

The primary task for configuring LoadLeveler Blue Gene support consists of setting up the environment of the **LoadL_negotiator** daemon, the environment of any process that will run Blue Gene jobs, and the LoadLeveler configuration file.

Perform the following steps to configure LoadLeveler Blue Gene support:

1. Configure the **LoadL_negotiator** daemon to run on a node which has access to the Blue Gene Control System.
2. Enable Blue Gene support by setting the **BG_ENABLED** configuration file keyword to **true.**
3. (Optional) Set any of the following additional Blue Gene related configuration file keywords which your setup requires:
   - **BG_ALLOW_LL_JOBS_ONLY**
   - **BG_CACHE_BLOCKS**
   - **BG_ENABLE_PASSTHROUGH**
   - **BG_MIN_BLOCK_SIZE**
   - **CM_CHECK_USERID**

   See "Configuration keyword descriptions" on page 233 for more information on these keywords.
4. Set any environment variables for the LoadL_negotiator daemon for Blue Gene. You can manually set the environment variable before starting LoadLeveler or set it into the global profile for LoadLeveler admin users.

   Note: Using the **llctl -h** or **llctl -g** command to start the central manager remotely will not carry the environment variables from the login session to the LoadLeveler daemons on the remote nodes.

   The following environment variables can be specified for Blue Gene:

   LL_BG_DRAIN_FILE =full/path/to/file
   Specify the full path name of the file containing midplanes that should not be used by LoadLeveler when scheduling jobs or reservations. The file must contain one entry per line and an entry can consist of an individual midplane or any part of the midplane location name.

   A file consisting of:

   ```
   # list the midplanes to be drained from the Blue Gene system
   R00-M0
   R1
   ```

   will drain R00-M0 and any midplanes starting with "R1" from the LoadLeveler cluster.

   Usage:

   ```
   export LL_BG_DRAIN_FILE=/bghome/loadl/drain_file
   ```

   BG_PROPERTIES_FILE=/full/path/to/file
   Specify the full path name of the **bg.properties** file which contains the data required to access the Blue Gene Control system. The default path used is **/bgsys/local/etc/bg.properties**.

   For details on the contents of the database property file, see *Blue Gene/Q: System Administration* on IBM Redbooks (http://www.redbooks.ibm.com/).

   Usage:

   ```
   export BG_PROPERTIES_FILE=/bgsys/local/etc/bg.properties
   ```

Blue Gene reservation support

Reservation supports Blue Gene resources including the Blue Gene compute nodes. It is important to note that when the reservation includes Blue Gene nodes, it cannot include conventional nodes. A front end node (FEN), which is used to start a Blue Gene job, is not part of the Blue Gene resources. A Blue Gene reservation only reserves Blue Gene resources and a Blue Gene job step bound to a reservation uses the reserved Blue Gene resources and shares a FEN outside the reservation.

Jobs using Blue Gene resources can be submitted to a Blue Gene reservation to run. A Blue Gene job step can also be used to select what Blue Gene resources to reserve to make sure the reservation will have enough Blue Gene resources to run the Blue Gene job step.

For more information about reservations, see "Overview of reservations" on page 24.

Blue Gene fair share scheduling support

Fair share scheduling has been extended to Blue Gene resources as well. The **FAIR_SHARE_TOTAL_SHARES** keyword in **LoadL_config** and the **fair_shares** keyword for the user and group stanza in **LoadL_admin** apply to both the CPU resources and the Blue Gene resources. When a Blue Gene job step ends, both the CPU utilization and the Blue Gene resource utilization data will be collected. The elapsed job running time multiplied by the number of C-nodes allocated to the job step (the **BG Size Allocated field** in the **llq -l** output) will be counted as the amount of Blue Gene resource used. The used shares of the Blue Gene resources are independent of the used shares of the CPU resources and are made available through the LoadLeveler variables **UserUsedBgShares** and **GroupUsedBgShares**. LoadLeveler variable **JobIsBlueGene** will indicate whether a job step is a Blue Gene job step or not. LoadLeveler administrators have flexibility in specifying the behavior of fair share scheduling by using these variables in the SYSPRIO expression. The **llfs** command and the related APIs can also handle requests related to the Blue Gene resources.

For more information about fair share scheduling, see "Using fair share scheduling" on page 158.

Blue Gene heterogeneous memory support

The LoadLeveler job command file has a **bg_requirements** keyword that can be used to specify the requirements that the Blue Gene midplanes must meet to execute the job step. The Blue Gene compute nodes (C-nodes) in the same midplane have the same amount of physical memory. The C-nodes in different midplanes might have different amounts of physical memory. The **bg_requirements** job command file keyword allows users to specify the memory requirement on the Blue Gene C-nodes.

The **bg_requirements** keyword works like the **requirements** keyword, but it can only support memory requirements and applies only to Blue Gene midplanes. For a Blue Gene job step, the **requirements** keyword value applies to the front end node needed by the job step and the **bg_requirements** keyword value applies to the Blue Gene midplanes needed by the job step.

Blue Gene preemption support

Preemption support for Blue Gene jobs has been enabled. Blue Gene jobs have the same preemption support as non-Blue Gene jobs. In a typical Blue Gene system,

many Blue Gene jobs share the same front end node while dedicated Blue Gene resources are used for each job. To avoid preempting Blue Gene jobs that use different Blue Gene resources as requested by a preempting job, **ENOUGH** instead of **ALL** must be used in the **PREEMPT_CLASS** rules for Blue Gene job preemption.

For more information about preemption, see "Preempting and resuming jobs" on page 122

Using fair share scheduling

Fair share scheduling in LoadLeveler provides a way to divide resources in a LoadLeveler cluster among users or groups of users.

To fairly share cluster resources, LoadLeveler can be configured to allocate a proportion of the resources to each user or group and to let job priorities be adjusted based on how much of the resources have been used and when they were used. Generally speaking, LoadLeveler should be configured so that job priorities decrease for a user or group that has recently used more resources than the allocated proportion and job priorities should increase for a user or group that has not run any jobs recently.

Administrators can configure the behavior of fair share scheduling through a set of configuration keywords. They can also query fair share information, save a snapshot of historic data, reset and restore fair share scheduling, and perform other functions by using the LoadLeveler **llfs** command and the corresponding APIs.

Fair share scheduling also includes Blue Gene resources (see "Blue Gene fair share scheduling support" on page 157 for more information).

Note: The time of day clocks on all of the nodes in the cluster *must* be synchronized in order for fair share scheduling to work properly.

For more information, see the following:

- **llfs** command (see *LoadLeveler: Command and API Reference*)
- Corresponding APIs:
  - **ll_fair_share** subroutine (see *LoadLeveler: Command and API Reference*)
  - Data access API (see *LoadLeveler: Command and API Reference*)
- Keywords:
  - fair_shares
  - FAIR_SHARE_INTERVAL
  - FAIR_SHARE_TOTAL_SHARES
- SYSPRIO expression

Fair share scheduling keywords

The **FAIR_SHARE_TOTAL_SHARES** global configuration file keyword is used to specify the total number of shares that each type of resource is divided into. The **fair_shares** keyword in a user or group stanza in the administration file specifies how many shares the user or group is allocated. The ratio of the **fair_shares** keyword value in a user or group stanza over the **FAIR_SHARE_TOTAL_SHARES** keyword value defines the resource usage proportion for the user or group. For example, if a user is allocated one third of

the cluster resources, then the ratio of the user's **fair_share** value over the **FAIR_SHARE_TOTAL_SHARES** keyword value should be one third.

The LoadLeveler SYSPRIO expression can be configured to let job priorities change to achieve the specified resource usage proportions. Besides changing job priorities, fair share scheduling does not change in any way how LoadLeveler schedules jobs. If a job can be scheduled to run, it will be run regardless of whether the owner and the LoadLeveler group of the job has any shares allocated or not. No matter how many shares are allocated to a user, if the user does not submit any jobs to run, then the resource usage proportion for that user cannot be achieved and other users might be able to use more than their allocated proportions.

Note: The sum of all allocated shares for users or groups does not have to equal the value of the **FAIR_SHARE_TOTAL_SHARES** keyword. The share allocation can be used as a way to prevent a single user from consuming too much of the cluster resources and as a way to share the resources as fairly as possible.

When the value of the **FAIR_SHARE_TOTAL_SHARES** keyword is greater than 0, fair share scheduling is on, which means that resource usage data is collected when every job ends, regardless of the **fair_shares** values for any user or group. The collected usage data is converted to used shares for each user and group. The **llfs** command can be used to display the allocated and used shares. Turning fair share scheduling on does not mean that job priorities are affected by fair share scheduling. You have to configure the SYSPRIO expression to let fair share scheduling affect job priorities in a way that suits your needs. By default, the value of the **FAIR_SHARE_TOTAL_SHARES** keyword is 0 and fair share scheduling is disabled.

There is a built-in decay mechanism for the historic resource usage data that is collected when jobs end, that is, the initial resource usage value becomes smaller and smaller as times goes by. This decay mechanism allows the most recent resource usage to have more impact on fair share scheduling. The **FAIR_SHARE_INTERVAL** global configuration file keyword is used to specify how fast the decay is. The shorter the interval, the faster the historic data decays. A resource usage value decays to 5% of its initial value after an elapsed time period of the same length as the **FAIR_SHARE_INTERVAL** value. Generally, the interval should be at least several times larger than the typical job running time in the cluster to get stable results. A value should be chosen corresponding to how long the historic resource usage data should have an impact on the current job priorities.

The LoadLeveler SYSPRIO expression is used to calculate job priorities. A set of LoadLeveler variables including some related to fair share scheduling can be used in the SYSPRIO expression in the global configuration file. You can define the SYSPRIO expression to let fair share scheduling influence the job priorities in a way that is suitable to your needs. For more information, see the SYSPRIO expression in Chapter 10, “Configuration keyword reference,” on page 231.

When the **GroupTotalShares**, **GroupUsedShares**, **UserTotalShares**, **UserUsedShares**, **UserUsedBgShares**, **GroupUsedBgShares**, and **JobIsBlueGene** and their corresponding user-defined variables are used, you must use the **NEGOTIATOR_RECALCULATE_SYSPRIO_INTERVAL** global configuration keyword to specify a time interval at which the job priorities will be recalculated using the most recent share usage information.

Note: These "UsedShares" variables are integer values (they are not floating point numbers). If greater precision is desired, increase the values for **FAIR_SHARE_TOTAL_SHARES** and **fair_shares** by some order of magnitude.

You can add the following user-defined variables to the **LoadL_config** global configuration file to make it easier to specify fair share scheduling in the SYSPRIO expressions:

- **GroupRemainingShares** = (GroupTotalShares - GroupUsedShares)
- **GroupHasShares** = ($(GroupRemainingShares) > 0)
- **GroupSharesExceeded** = ($(GroupRemainingShares) <= 0)
- **UserRemainingShares** = (UserTotalShares - UserUsedShares)
- **UserHasShares** = ($(UserRemainingShares) > 0)
- **UserSharesExceeded** = ($(UserRemainingShares) <= 0)
- **UserRemainingBgShares** = (UserTotalShares - UserUsedBgShares)
- **UserHasBgShares** = ($(UserRemainingBgShares) > 0)
- **UserBgSharesExceeded** = ($(UserRemainingBgShares) <= 0)
- **GroupRemainingBgShares** = (GroupTotalShares - GroupUsedBgShares)
- **GroupHasBgShares** = ($(GroupRemainingBgShares) > 0)
- **GroupBgSharesExceeded** = ($(GroupRemainingBgShares) <= 0)
- **JobIsNotBlueGene** = ! JobIsBlueGene

If fair share scheduling is not turned on, either because the **FAIR_SHARE_INTERVAL** keyword value is not positive or because the scheduler type is not **BACKFILL**, then the variables will have the following values:

```
GroupTotalShares: 0
GroupUsedShares: 0
$(GroupRemainingShares): 0
$(GroupHasShares): 0
$(GroupSharesExceeded): 1
UserUsedBgShares: 0
$(UserRemainingBgShares): 0
$(UserHasBgShares): 0
$(UserBgSharesExceeded): 1
```

If a user has the **fair_shares** keyword set to 10 in its user stanza and the user has used up 8 CPU shares and 3 Blue Gene shares, then the variables will have the following values:

```
UserTotalShares: 10
UserUsedShares: 8
$(UserRemainingShares): 2
$(UserHasShares): 1
$(UserSharesExceeded): 0
UserUsedBgShares: 3
$(UserRemainingBgShares): 7
$(UserHasBgShares): 1
$(UserBgSharesExceeded): 0
```

If a group has the **fair_shares** keyword set to 10 in its group stanza and the group has used up 15 CPU shares and 0 Blue Gene shares, then the variables will have the following values:

```
GroupTotalShares: 10
GroupUsedShares: 15
$(GroupRemainingShares): -5
$(GroupHasShares): 0
$(GroupSharesExceeded): 1
GroupUsedBgShares: 0
$(GroupRemainingBgShares): 10
$(GroupHasBgShares): 1
$(GroupBgSharesExceeded): 0
```

The values of the following variables for a Blue Gene job step:

```
JobIsBlueGene: 1
$(JobIsNotBlueGene): 0
```

The values of the following variables for a non-Blue Gene job step:

```
JobIsBlueGene: 0
$(JobIsNotBlueGene): 1
```

Reconfiguring fair share scheduling keywords

LoadLeveler configuration and administration files can be modified to assign new values to various keywords. After files have been modified, issue the **llctl -g reconfig** command to read in the new keyword values. All new keywords introduced for fair share scheduling become effective right after reconfiguration.

Reconfiguring when the Schedd daemons are up

To avoid any inconsistency, change the value of the **FAIR_SHARE_INTERVAL** keyword while the central manager and all **Schedd** daemons are up, then do the reconfiguration. After the reconfiguration, the following will happen:

- All historic fair share scheduling data will be decayed to the current time using the old value.
- The old value is replaced with the new value
- The new value will be used from here on

Note:

1. You must have the same value for the **FAIR_SHARE_INTERVAL** keyword in the central manager and the **Schedd** daemons because the **FAIR_SHARE_INTERVAL** keyword determines the rate of decay for the historic fair share data and the same value on the daemons maintains the data consistency.
2. There are some LoadLeveler configuration parameters that require restarting LoadLeveler with **llctl recycle** for changes to take effect. You can use **llctl recycle** when changing fair share parameters also. The effect will be the same as using **llctl reconfig** because when the Schedd machine shuts down normally, the fair share scheduling data will be decayed to the time of the shutdown and it will be saved.

Reconfiguring when the Schedd daemons are down

The value for the **FAIR_SHARE_INTERVAL** keyword may need to be changed while a **Schedd** daemon is down.

If the value for the **FAIR_SHARE_INTERVAL** keyword has to be changed while a **Schedd** daemon is down, the following will happen when the **Schedd** daemon is restarted:

- All historic fair share scheduling data will be read in from the disk files in the $(SPOOL) directory with no change.
- When a new job ends, the historic fair share scheduling data for the owner and the LoadLeveler group of the job will be updated using the new value and then sent to the central manager. The new value is used effectively from the time the data was last updated before the **Schedd** went down, not from the time of the reconfiguration as it would normally be.

Example: three groups share a LoadLeveler cluster

For purposes of this example, we will assume the following:

- Three groups of users share a LoadLeveler cluster and each group is to have one third of the resources
- Historic data will have significant impact for about 10 days
- Groups with unused shares will have much higher job priorities than the groups which have used up their shares

To setup for fair share scheduling with these assumptions, an administrator could update the **LoadL_config** global configuration file as follows:

```
FAIR_SHARE_TOTAL_SHARES = 99

FAIR_SHARE_INTERVAL = 240

NEGOTIATOR_RECALCULATE_SYSPRIO_INTERVAL = 300

GroupRemainingShares = ( GroupTotalShares - GroupUsedShares )

GroupHasShares = ( $(GroupRemainingShares) > 0 )

SYSPRIO : 10000000 * $(GroupHasShares) - QDate
```

In the admin file **LoadL_admin**, add:

```
chemistry: type = group

   include_users = harold mark kim enci george charlie

   fair_shares = 33

physics: type = group

   include_users = cnyang gchen newton roy

   fair_shares = 33

math: type = group

   include_users = rich dave chris popco
   fair_shares = 33
```

When user rich in the math group wants to submit a job, the following keyword can be put into the job command file so that the job will have high priority through the math group:

```
#@group=math
```

If user rich has a job that does not need to be run right away or as soon as possible (can be run at any time), then he should run the job in a LoadLeveler group with no shares allocated (for example, the **No_Group** group). Because the group **No_Group** has no shares allocated to it in this example, $(GroupHasShares) has a value of 0 and the job priority will be lower than those jobs whose group has unused shares. The job will be run when all higher priority jobs are done or when it is used to backfill a higher priority job (will be run whenever it can be scheduled).

Example: two thousand students share a LoadLeveler cluster

For purposes of this example, we will assume the following:

- A university has 2000 students who share a LoadLeveler cluster and every student is to have the same number of shares of the resources.
- Historic data will have significant impact for about 7 days (because **FAIR_SHARE_INTERVAL** is not specified and the default value is 7 days).

- A student with unused shares is to have somewhat higher job priorities and let the priorities decrease as the number of used shares increase.

The **LoadL_config** global configuration file should contain the following:

```
FAIR_SHARE_TOTAL_SHARES = 10000

NEGOTIATOR_RECALCULATE_SYSPRIO_INTERVAL = 600

UserRemainingShares = ( UserTotalShares - UserUsedShares )

SYSPRIO : 100000 * $(UserRemainingShares) - QDate
```

In the **LoadL_admin** admin file, add

```
default: type = user

  fair_shares = 5
```

Note: The value **fair_shares = 5** is the result of dividing the total shares into the number of students (10000 ÷ 2000). The number of students can be more or less than 2000, but the same configuration parameters still prevent a single user from using too much cluster resources in a short time period.

We can see from the SYSPRIO expression that the larger the number of unused shares for a student and the earlier the job is submitted, the higher the priority is for the student's job.

Querying information about fair share scheduling

The **llfs** command and the data access API can be used to query information about fair share scheduling. The **llfs** command without any options displays the allocated and used shares for all users and LoadLeveler groups having run one or more jobs in the cluster to completion. The **-u** and **-g** options can show the allocated and used shares for any user or LoadLeveler group regardless of whether they have run any jobs in the cluster. In either case, the user or group need not have any **fair_shares** allocated in the **LoadL_admin** administration file for the usage to be reported by the **llfs** command.

Resetting fair share scheduling

The **llfs -r** command option, by default, will start fair share scheduling from the beginning, which means that all the previous historic data will be lost. This command will not be run unless all **Schedd** daemons are up and running.

In case a **Schedd** daemon is down when this command option is being run, the request will not be processed. To manually reset fair share scheduling, bring down the LoadLeveler cluster, remove all fair share data files (**fair_share_queue.dir** and **fair_share_queue.pag**) in the $(SPOOL) directory and then restart the LoadLeveler cluster.

Saving historic data

The LoadLeveler central manager holds the complete historic fair share data when it is up. Every Schedd holds a portion of the historic fair share data and the data is stored on disk in the $(SPOOL) directory. When the central manager is restarted, it receives the historic fair share data from every Schedd. If a Schedd machine is down temporarily and the central manager remains up, the data in the central manager is not affected. In case a Schedd machine is permanently damaged and the central manager restarts, the central manager will not be able to get all of the historic fair share data because the data stored on the damaged Schedd is lost. If

the value of **FAIR_SHARE_INTERVAL** is very large, many days of data on the damaged Schedd could be lost. To reduce the loss of data, the historic fair share data in the central manager can be saved to disk periodically. Recovery can be done using the latest saved data when a Schedd machine is permanently out of service. The **llfs -s** command or the **ll_fair_share** API can be used to save a snapshot of the historic data in the central manager to a file.

Restoring saved historic data

You can use the **llfs -r** command option or the **ll_fair_share** API to restore fair share scheduling to a previously saved state. For the file name, specify a file you saved previously using **llfs -s**.

If the central manager goes down and restarts again, the historic data stored in an out of service Schedd machine is not reported to the central manager. If the Schedd machine will not be brought back to service at all, then the administrator can consider restoring fair share scheduling to a state corresponding to the latest saved file.

Procedure for recovering a job spool

The **llmovespool** command is intended for recovery purposes only.

Jobs being managed by a down Schedd are unable to clean up resources or move to completion. These jobs need their job records transferred to another Schedd. The **llmovespool** command moves the job records from the spool of one managing Schedd to another managing Schedd in the local cluster. All moved jobs retain their original job identifiers.

It is very important that the Schedd that created the job records to be moved is not running during the move operation. Jobs within the job queue database will be unrecoverable if the job queue is updated during the move by any process other than the **llmovespool** command.

The **llmovespool** command operates on a set of job records, these records are updated as the command executes. When a job is successfully moved, the records for that job are deleted. Job records that are not moved because of a recoverable failure, like the original Schedd not being fenced, may have the **llmovespool** command executed against them again. It is very important that a Schedd never reads the job records from the spool being moved. Jobs will be unrecoverable if more than one Schedd is considered to be the managing Schedd.

The procedure for recovering a job spool is:

1. Move the files located in the spool directory to be transferred to another directory before entering the **llmovespool** command in order to guarantee that no other Schedd process is updating the job records.
2. Add the statement **schedd_fenced=true** to the machine stanza of the original Schedd node in order to guarantee that the central manager ignores connections from the original managing Schedd, and to prevent conflicts from arising if the original Schedd is restarted after the **llmovespool** command has been run. See the **schedd_fenced=true** keyword in Chapter 11, "Administration keyword reference," on page 293 for more information.
3. Reconfigure the central manager node so that it recognizes that the original Schedd is "fenced".

4. Issue the **llmovespool** command providing the spool directory where the job records are stored. The command displays a message that the transfer has started and reports status for each job as it is processed. For more information about the **llmovespool** command and the **ll_move_spool** API, see *LoadLeveler: Command and API Reference*.

Configuring and using island scheduling

The **island** keyword allows you to specify the name of the island a machine or machine_group belongs to.

Many large clusters have a network topology where groups of nodes have greater bandwidth for communicating among themselves than they do when communicating with nodes outside the group. These groups are often referred to as *islands* and so that is the term used by LoadLeveler. These islands could be based on other characteristics like the drawer or frame that the node belongs to and the concept is the same.

The **island** keyword allows you to specify the name of the island a machine or machine_group belongs to. Each machine can belong to only one island.

To request island topology, use the **node_topology** job command file keyword. When you specify **node_topology = island**, you can also specify a second job command file keyword to control how many islands are selected. For example:

```
island_count=number1 [,number2]
```

If the **island_count** keyword is not specified, the default behavior is for all selected machines for the job to be from a common island.

Optionally, you can also use the existing **requirements** or **preferences** job command file keywords to request specific islands by name.

For more information about any of these job command keywords, see "Job command file keyword descriptions" on page 335.

When scheduling a job step with an island topology requirement, the LoadLeveler scheduler will select the island where the job step best "fits." For example, if a job step requires 32 nodes, and there is one island with 48 nodes available and another island with 512 nodes available, the scheduler will allocate machines from the first island. If the next job step also asks for 32 nodes, then the machines will be taken from the second island, because only 16 nodes would remain in the first island, which is insufficient for this job step.

The reasoning behind this mode of island selection is to leave an unused island unused for as long as possible so that when a larger job is submitted to the cluster, there remains an island with enough available machines to accommodate that larger job.

With a requirement for only one island, it is simple: LoadLeveler will fit the job step onto the island with the lowest number of available machines.

Note: The machines selected within each island are selected based on their **MACHPRIO** order, just as machines would be selected for a job step that does not have an island topology requirement.

When the requirement is for more than one island, LoadLeveler will attempt to distribute the job step evenly across all islands used. For example, if the job step requires 1024 nodes and has **island_count=8,2**, then LoadLeveler will try to allocate 128 nodes from each of 8 islands. It is possible that only 6 of the islands have 128 or more nodes free, while 2 other islands have fewer than 128 nodes free. LoadLeveler will still use machines from 8 islands because of the **island_count** request. All remaining nodes from those last 2 islands will be used, and the remaining tasks will be divided as evenly as possible among the other 6 islands.

Suppose instead that the same 1024-node job with **island_count=8,2** is submitted at a time when only 6 islands have any machines available at all. Because LoadLeveler can use as few as 2 islands, it is permissible to use just 6 islands for the job. If any of those 6 islands have fewer than 171 (1024/6=170 with a remainder of 4) nodes available, then all of the available nodes in that island will be used and the remaining tasks will be divided as evenly as possible among the remaining available islands.

And in another scenario with the same 1024-node job, suppose there are 10 islands each having 128 or more nodes free and another 2 islands with fewer than 128 nodes free. LoadLeveler will select 128 nodes from each of 8 of those 10 islands and will not select any nodes from the 2 islands that do not have 128 nodes available. It is more important to satisfy the user's request for the job of running across 8 islands than to try to fill up the remaining nodes on islands which are almost full.

Energy aware job support

Using the energy function, a job can run with a lower CPU frequency to save energy.

You can set an acceptable performance degradation (**max_perf_decrease_allowed**) or required energy saving (**energy_saving_req**) for the job in the job command file. LoadLeveler will choose a suitable CPU frequency for the job or reject its submission based on the specified value.

The energy function requires database support.

See the following:

- *LoadLeveler for Linux: Installation Guide* for information about setting up the optional LoadLeveler energy function
- "Working with energy aware jobs" on page 220 for the steps on how to use the energy keywords in the job command file

S3 state support

The S3 state refers to a standby state where RAM remains powered. S3 state support allows the system administrator to switch an idle node to standby state to save energy. LoadLeveler provides a time-based policy to the administrator to decide the start time and duration for the idle node to enter S3 state. The policy is configured for machines or machine_group stanzas using the keyword **power_management_policy**. After the administrator sets up the policy for the compute node, LoadLeveler checks the policy and switches the state of the node automatically at the specified policy start time. The node will switch back to

working state (S0) after that time frame has elapsed. An action to take if a machine fails to change to the standby state can be set using the configuration keyword **SUSPEND_CONTROL**.

The CPU does not execute instructions in standby (S3) state, so the node cannot resume from S3 state. The idle node state change is initiated by the resource manager daemon. LoadLeveler calls the xCAT **rpower** command to accomplish the state change.

Part 3. Submitting and managing LoadLeveler jobs

After an administrator installs IBM LoadLeveler and customizes the environment, general users can build and submit jobs to exploit the many features of the LoadLeveler runtime environment.

Chapter 7. Building and submitting jobs

Learn more about building and submitting jobs.

The topics listed Table 36 will help you learn about building and submitting jobs:

Table 36. Learning about building and submitting jobs

To learn about:	Read the following:
Creating and submitting serial and parallel jobs	Chapter 7, "Building and submitting jobs"
Controlling and monitoring LoadLeveler jobs	Chapter 8, "Managing submitted jobs," on page 223
Ways to control or monitor LoadLeveler operations by using the LoadLeveler commands and APIs	• *LoadLeveler: Command and API Reference* • Chapter 9, "Example: Using commands to build, submit, and manage jobs," on page 227

Table 37 lists the tasks that general users perform to run LoadLeveler jobs.

Table 37. Roadmap of user tasks for building and submitting jobs

To learn about:	Read the following:
Building jobs	• "Building a job command file" • "Editing job command files" on page 176 • "Defining resources for a job step" on page 177 • "Working with coscheduled job steps" on page 178 • "Using bulk data transfer" on page 180 • "Preparing a job for checkpoint/restart" on page 180 • "Preparing a job for preemption" on page 183
Submitting jobs	• "Submitting a job command file" on page 183 • **llsubmit** command (see *LoadLeveler: Command and API Reference*)
Working with parallel jobs	"Working with parallel jobs" on page 184
Working with reserved node resources and the jobs that use them	"Working with reservations" on page 203
Correctly specifying job command file keywords	Chapter 12, "Job command file reference," on page 333

Building a job command file

Before you can submit a job or perform any other job related tasks, you need to build a job command file.

A job command file describes the job you want to submit, and can include LoadLeveler keyword statements. For example, to specify a binary to be executed, you can use the **executable** keyword, which is described later in this topic. To specify a shell script to be executed, the **executable** keyword can be used; if it is not used, LoadLeveler assumes that the job command file itself is the executable.

The job command file can include the following:

- LoadLeveler keyword statements: A *keyword* is a word that can appear in job command files. A *keyword statement* is a statement that begins with a LoadLeveler keyword. These keywords are described in "Job command file keyword descriptions" on page 335.
- Comment statements: You can use comments to document your job command files. You can add comment lines to the file as you would in a shell script.
- Shell command statements: If you use a shell script as the executable, the job command file can include shell commands.
- LoadLeveler variables: See "Job command file variables" on page 383 for more information.

You can use any text editor to build a job command file.

Using multiple steps in a job command file

To specify a stream of job steps, you need to list each job step in the job command file. You must specify one **queue** statement for each job step. Also, the executables for all job steps in the job command file must exist when you submit the job. For most keywords, if you specify the keyword in a job step of a multi-step job, its value is inherited by all proceeding job steps. Exceptions to this are noted in the keyword description.

LoadLeveler treats all job steps as independent job steps unless you use the **dependency** keyword. If you use the **dependency** keyword, LoadLeveler determines whether a job step should run based upon the exit status of the previously run job step.

For example, Figure 12 contains two separate job steps. Notice that `step1` is the first job step to run and that `step2` is a job step that runs only if `step1` exits with the correct exit status.

```
# This job command file lists two job steps called "step1"
# and "step2".  "step2" only runs if "step1" completes
# with exit status = 0.  Each job step requires a new
# queue statement.
#
# @ step_name = step1
# @ executable = executable1
# @ input = step1.in1
# @ output = step1.out1
# @ error = step2.err1
# @ queue
# @ dependency = (step1 == 0)
# @ step_name = step2
# @ executable = executable2
# @ input = step2.in1
# @ output = step2.out1
# @ error = step2.err1
# @ queue
```

Figure 12. Job command file with multiple steps

In Figure 12, `step1` is called the *sustaining* job step. `step2` is called the *dependent* job step because whether or not it begins to run is dependent upon the exit status of `step1`. A single sustaining job step can have more than one dependent job steps and a dependent job step can also have job steps dependent upon it.

In Figure 12 on page 172, each job step has its own **executable**, **input**, **output**, and **error** statements. Your job steps can have their own separate statements, or they can use those statements defined in a previous job step. For example, in Figure 13, `step2` uses the **executable** statement defined in `step1`:

```
# This job command file uses only one executable for
# both job steps.
#
# @ step_name = step1
# @ executable = executable1
# @ input = step1.in1
# @ output = step1.out1
# @ error = step1.err1
# @ queue
# @ dependency = (step1 == 0)
# @ step_name = step2
# @ input = step2.in1
# @ output = step2.out1
# @ error = step2.err1
# @ queue
```

Figure 13. Job command file with multiple steps and one executable

Examples: Job command files

These examples of job command files may apply to your situation.

- **Example 1: Generating multiple jobs with varying outputs**

  To run a program several times, varying the initial conditions each time, you could can multiple LoadLeveler scripts, each specifying a different input and output file as described in Figure 15 on page 175. It would probably be more convenient to prepare different input files and submit the job only once, letting LoadLeveler generate the output files and do the multiple submissions for you.

  Figure 14 illustrates the following:

  – You can refer to the LoadLeveler name of your job symbolically, using **$(jobid)** and **$(stepid)** in the LoadLeveler script file.

  – **$(jobid)** refers to the job identifier.

  – **$(stepid)** refers to the job step identifier and increases after each **queue** command. Therefore, you only need to specify input, output, and error statements once to have LoadLeveler name these files correctly.

  Assume that you created five input files and each input file has different initial conditions for the program. The names of the input files are in the form **longjob.in.***x*, where *x* is 0–4.

  Submitting the LoadLeveler script shown in Figure 14 results in your program running five times, each time with a different input file. LoadLeveler generates the output file from the LoadLeveler job step IDs. This ensures that the results from the different submissions are not merged.

```
# @ executable = longjob
# @ input = longjob.in.$(stepid)
# @ output = longjob.out.$(jobid).$(stepid)
# @ error = longjob.err.$(jobid).$(stepid)
# @ queue
# @ queue
# @ queue
# @ queue
# @ queue
```

Figure 14. Job command file with varying input statements

To submit the job, type the command:

```
llsubmit longjob.cmd
```

LoadLeveler responds by issuing the following:

```
submit: The job "ll6.23" with 5 job steps has been submitted.
```

Table 38 lists the standard input files, standard output files, and standard error files for the five job steps:

Table 38. Standard files for the five job steps

Job Step	Standard Input	Standard Output	Standard Error
ll6.23.0	longjob.in.0	longjob.out.23.0	longjob.err.23.0
ll6.23.1	longjob.in.1	longjob.out.23.1	longjob.err.23.1
ll6.23.2	longjob.in.2	longjob.out.23.2	longjob.err.23.2
ll6.23.3	longjob.in.3	longjob.out.23.3	longjob.err.23.3
ll6.23.4	longjob.in.4	longjob.out.23.4	longjob.err.23.4

- **Example 2: Using LoadLeveler variables in a job command file**

  Figure 15 on page 175 shows how you can use LoadLeveler variables in a job command file to assign different names to input and output files. This example assumes the following:

  – The name of the machine from which the job is submitted is `lltest1`
  – The user's home directory is **/u/rhclark** and the current working directory is **/u/rhclark/OSL**
  – LoadLeveler assigns a value of 122 to **$(jobid)**.

  In Job Step 0:

  – LoadLeveler creates the subdirectories **oslsslv_out** and **oslsslv_err** if they do not exist at the time the job step is started.

  In Job Step 1:

  – The character string **rhclark** denotes the home directory of user **rhclark** in **input**, **output**, **error**, and **executable** statements.
  – The **$(base_executable)** variable is set to be the "base" portion of the **executable**, which is **oslsslv**.
  – The **$(host)** variable is equivalent to **$(hostname)**. Similarly, **$(jobid)** and **$(stepid)** are equivalent to **$(cluster)** and **$(process)**, respectively.

  In Job Step 2:

  – This job step is executed only if the return codes from Step 0 and Step 1 are both equal to zero.
  – The initial working directory for Step 2 is explicitly specified.

```
# Job step 0 ===========================================================
#   The names of the output and error files created by this job step are:
#
#     output: /u/rhclark/OSL/oslsslv_out/lltest1.122.0.out
#     error : /u/rhclark/OSL/oslsslv_err/lltest1_122_0_err
#
# @ job_name = OSL
# @ step_name = step_0
# @ executable = oslsslv
# @ arguments = -maxmin=min -scale=yes -alg=dual
# @ environment = OSL_ENV1=20000; OSL_ENV2=500000
# @ requirements = (Arch == "R6000") && (OpSys == "AIX71")
# @ input  = test01.mps.$(stepid)
# @ output = $(executable)_out/$(host).$(jobid).$(stepid).out
# @ error  = $(executable)_err/$(host)_$(jobid)_$(stepid)_err
# @ queue
#
# Job step 1 ===========================================================
#   The names of the output and error files created by this job step are:
#
#     output: /u/rhclark/OSL/oslsslv_out/lltest1.122.1.out
#     error : /u/rhclark/OSL/oslsslv_err/lltest1_122_1_err
#
# @ step_name = step_1
# @ executable = rhclark/$(job_name)/oslsslv
# @ arguments = -maxmin=max -scale=no -alg=primal
# @ environment = OSL_ENV1=60000; OSL_ENV2=500000; \
                  OSL_ENV3=70000; OSL_ENV4=800000;
# @ input  = rhclark/$(job_name)/test01.mps.$(stepid)
# @ output = rhclark/$(job_name)/$(base_executable)_out/$(hostname).$(cluster).$(process).out
# @ error  = rhclark/$(job_name)/$(base_executable)_err/$(hostname)_$(cluster)_$(process)_err
# @ queue
#
# Job step 2 ===========================================================
#   The names of the output and error files created by this job step are:
#
#     output: /u/rhclark/OSL/oslsslv_out/lltest1.122.2.out
#     error : /u/rhclark/OSL/oslsslv_err/lltest1_122_2_err
#
# @ step_name = OSL
# @ dependency = (step_0 == 0) && (step_1 == 0)
# @ comment = oslsslv
# @ initialdir = /u/rhclark/$(step_name)
# @ arguments = -maxmin=min -scale=yes -alg=dual
# @ environment = OSL_ENV1=300000; OSL_ENV2=500000
# @ input  = test01.mps.$(stepid)
# @ output = $(comment)_out/$(host).$(jobid).$(stepid).out
# @ error  = $(comment)_err/$(host)_$(jobid)_$(stepid)_err
# @ queue
```

Figure 15. Using LoadLeveler variables in a job command file

- **Example 3: Using the job command file as the executable**

  The name of the sample script shown in Figure 16 on page 176 is **run_spice_job**. This script illustrates the following:

  – The script does not contain the **executable** keyword. When you do not use this keyword, LoadLeveler assumes that the script is the executable. (Since the name of the script is **run_spice_job**, you can add the **executable = run_spice_job** statement to the script, but it is not necessary.)

  – The job consists of four job steps (there are 4 **queue** statements). The **spice3f5** and **spice2g6** programs are invoked at each job step using different input data files:

    - **spice3f5:** Input for this program is from the file **spice3f5_input**_*x* where *x* has a value of 0, 1, and 2 for job steps 0, 1, and 2, respectively. The name of this file is passed as the first argument to the script. Standard output and standard error data generated by **spice3f5** are directed to the file **spice3f5_output**_*x*. The name of this file is passed as second argument to the script. In job step 3, the names of the input and output files are **spice3f5_input_benchmark1** and **spice3f5_output_benchmark1**, respectively.

- **spice2g6:** Input for this program is from the file **spice2g6_input**_*x*. Standard output and standard error data generated by **spice2g6** together with all other standard output and standard error data generated by this script are directed to the files **spice_test_output**_*x* and **spice_test_error**_*x*, respectively. In job step 3, the name of the input file is **spice2g6_input_benchmark1**. The standard output and standard error files are **spice_test_output_benchmark1** and **spice_test_error_benchmark1**.

All file names that are not fully qualified are relative to the initial working directory **/home/loadl/spice**. LoadLeveler will send the job steps 0 and 1 of this job to a machine for that has a real memory of 64 MB or more for execution. Job step 2 most likely will be sent to a machine that has more that 128 MB of real memory and has the ESSL library installed since these preferences have been stated using the LoadLeveler **preferences** keyword. LoadLeveler will send job step 3 to the machine `ll5.pok.ibm.com` for execution because of the explicit requirement for this machine in the **requirements** statement.

```
#!/bin/ksh
# @ job_name = spice_test
# @ account_no = 99999
# @ class = small
# @ arguments = spice3f5_input_$(stepid) spice3f5_output_$(stepid)
# @ input  = spice2g6_input_$(stepid)
# @ output = $(job_name)_output_$(stepid)
# @ error  = $(job_name)_error_$(stepid)
# @ initialdir = /home/loadl/spice
# @ requirements = ((Arch == "R6000") && \
#           (OpSys == "AIX71") && (Memory > 64))
# @ queue
# @ queue
# @ preferences  = ((Memory > 128)  && (Feature == "ESSL"))
# @ queue
# @ class = large
# @ arguments = spice3f5_input_benchmark1 spice3f5_output_benchmark1
# @ requirements = (Machine == "ll5.pok.ibm.com")
# @ input  = spice2g6_input_benchmark1
# @ output = $(job_name)_output_benchmark1
# @ error  = $(job_name)_error_benchmark1
# @ queue
OS_NAME='uname'

case $OS_NAME in
   AIX)
      echo "Running $OS_NAME version of spice3f5" > $2
      AIX_bin/spice3f5 < $1  >> $2  2>&1
      echo "Running $OS_NAME version of spice2g6"
      AIX_bin/spice2g6
      ;;
   *)
      echo "spice3f5 for $OS_NAME is not available" > $2
      echo "spice2g6 for $OS_NAME is not available"
      ;;
esac
```

Figure 16. Job command file used as the executable

Editing job command files

After you build a job command file, you can edit it using the editor of your choice.

You may want to change the name of the executable or add or delete some statements.

When you create a job command file, it is considered the job executable unless you specify otherwise by using the **executable** keyword in the job command file. LoadLeveler copies the executable to the spool directory unless the **checkpoint** keyword was set to **yes** or **interval**. Jobs that are to be checkpointed cannot be moved to the spool directory. Do not make any changes to the executable while the job is still in the queue–it could affect the way that job runs.

Defining resources for a job step

The LoadLeveler user may use the **resources** keyword in the job command file to specify the resources to be consumed by each task of a job step.

If the **resources** keyword is specified in the job command file, it overrides any **default_resources** specified by the administrator for the job step's class.

For example, the following job requests one CPU and one FRM license for each of its tasks:

```
resources = ConsumableCpus(1) FRMlicense(1)
```

If this were specified in a serial job step, one CPU and one FRM license would be consumed while the job step runs. If this were a parallel job step, then the number of CPUs and FRM licenses consumed while the job step runs would depend upon how many tasks were running on each machine. For more information on assigning tasks to nodes, see "Task-assignment considerations" on page 186.

Alternatively, you can use the **node_resources** keyword in the job command file to specify the resources to be consumed by the job step on each machine it runs on, regardless of the number of tasks assigned to each machine. If the **node_resources** keyword is specified in the job command file, it overrides the **default_node_resources** specified by the administrator for the job step's class.

For example, the following job requests 240 MB of **ConsumableMemory** on each machine:

```
node_resources = ConsumableMemory(240 mb)
```

Even if one machine only runs one task of the job step, while other machines run multiple tasks, 240 MB will be consumed on every machine.

You can use the **step_resources** keyword in the job command file to specify the floating resources to be consumed by the job step as a whole (as opposed to the task-based **resources** or the node-based **node_resources**).

For example, the following job step requests 100 units of storage:

```
step_resources = storage(100)
```

Submitting jobs requesting data staging

The **dstg_in_script** keyword causes LoadLeveler to generate an inbound data staging step, without requiring the #@queue specification. The value assigned to this keyword is the executable that will be started for data staging and any arguments needed by this script or executable as well.

The **dstg_in_wall_clock_limit** keyword specifies a wall clock time for the inbound data staging step. Specifying the estimated wall clock limit is mandatory when a data staging script is specified. Similarly, **dstg_out_script** and

dstg_out_wall_clock_limit will be used for generation and execution of the outbound data staging step for the job. All data staging job steps are assigned to the predefined class called **data_stage**.

Resources required for data staging can be specified using the **dstg_resources** keyword.

The **dstg_node** keyword allows you to specify how data replicas must be created:

- If the value specified is **any**, one data staging task is executed on any available node in the cluster with data staging resources. This value can be used with either the **at_submit** or the **just_in_time** configuration options.
- If the value specified is **master**, one data staging task is executed on the master node. The master node is the machine that will be used to run the inbound and outbound data staging steps as well as the first application step of the job.
- If the value is **all**, a data staging task is executed on each of the nodes that will be or were used by the first application step.

Any environment variables needed by the data staging scripts can be specified using the **dstg_environment** keyword. The **copy_all** value can be assigned to this keyword to get all of the user's environment variables.

For detailed information about the data staging job command file keywords, see "Job command file keyword descriptions" on page 335.

Working with coscheduled job steps

LoadLeveler allows you to specify that a group of two or more steps within a job are to be coscheduled. Coscheduled steps are dispatched at the same time.

Submitting coscheduled job steps

The **coschedule = yes** keyword in the job command file is used to specify which steps within a job are to be coscheduled. All steps within a job with the **coschedule** keyword set to **yes** will be coscheduled. The coscheduled steps will continue to be stored as individual steps in both memory and in the job queue, but when performing certain operations, such as scheduling, the steps will be managed as a single entity. An operation initiated on one of the coscheduled steps will cause the operation to be performed on all other steps (unless the coscheduling dependency between steps is broken).

Determining priority for coscheduled job steps

Coscheduled steps are supported only with the BACKFILL scheduler. The LoadLeveler BACKFILL scheduler will only dispatch the set of coscheduled steps when enough resource is available for all steps in the set to start. If the set of coscheduled steps cannot be started immediately, but enough resource will be available in the future, then the resource for all the steps will be reserved. In this case, only one of the coscheduled steps will be designated as a top dog, but enough resources will be reserved for all coscheduled steps and all the steps will be dispatched when the top dog step is started. The coscheduled step with the highest priority in the current job queue will be designated as the primary coscheduled step and all other steps will be secondary coscheduled steps. The primary coscheduled step will determine when the set of coscheduled steps will be scheduled. The priority for all other coscheduled steps is ignored.

Supporting preemption of coscheduled job steps

Preemption of coscheduled steps is supported with the following restrictions:

- In order for a step S to be preemptable by a coscheduled step, all steps in the set of coscheduled steps must be able to preempt step S.
- In order for a step S to preempt a coscheduled step, all steps in the set of coscheduled steps must be preemptable by step S.
- The set of job steps available for preemption will be the same for all coscheduled steps. Any resource made available by preemption for one coscheduled step will be available to all other coscheduled steps.

To determine the preempt type and preempt method to use when a coscheduled step preempts another step, an order of precedence for preempt types and preempt methods has been defined. All steps in the preempting coscheduled step are examined and the preempt type and preempt method having the highest precedence are used. The order of precedence for preempt type will be **ALL** and **ENOUGH**. The precedence order for preempt method is:

- Remove
- Vacate
- System Hold
- User hold
- Suspend

For more information about preempt types and methods, see "Planning to preempt jobs" on page 124.

When coscheduled steps are running, if one step is preempted as a result of a system-initiated preemption, then all coscheduled steps are preempted. When determining an optimal preempt set, the BACKFILL scheduler does not consider coscheduled steps as a single entity. All coscheduled steps are in the initial preempt set, but the final preempt set might not include all coscheduled steps, if the scheduler determines the resources of some coscheduled steps are not necessary to start the preempting job step. This implies that more resource than necessary might be preempted when a coscheduled step is in the set of steps to be preempted because regardless of whether or not all coscheduled steps are in the preempt set, if one coscheduled step is preempted, then all coscheduled steps will be preempted.

Coscheduled job steps and commands and APIs

Commands and APIs that operate on job steps are impacted by coscheduled steps. For the **llbind**, **llcancel**, **llhold**, and **llpreempt** commands, even if all coscheduled steps are not in the list of targeted steps, the requested operation is performed on all coscheduled steps.

For the **llmkres** and **llchres** commands, a coscheduled job step cannot be specified when using the **-j** or **-f** flags. For the **llckpt** command, you cannot specify a coscheduled job step using the **-u** flag.

Termination of coscheduled steps

If a coscheduled step is dispatched but cannot be started and is rejected by the **startd** daemon or the starter process, then all coscheduled steps are rejected. If a running step is removed or vacated by LoadLeveler as a result of a system related

failure, then all coscheduled steps are removed or vacated. If a running step is vacated as a result of the VACATE expression evaluating to true for the step, then all coscheduled steps are vacated.

Using bulk data transfer

On systems with device drivers and network adapters that support remote direct-memory access (RDMA), LoadLeveler supports bulk data transfer for jobs that use either the Internet or user space communication protocol mode.

You do not need to perform specific job-definition tasks to enable bulk transfer for LoadLeveler jobs that use the IP network protocol. LoadLeveler cannot affect whether IP communication uses bulk transfer; the implementation of IP where the job runs determines whether bulk transfer is supported.

To enable user space jobs to use bulk data transfer, however, **all** of the following tasks must be completed. If you omit one or more of these steps, the job will run but will not be able to use bulk transfer.

- Users must request bulk transfer for their LoadLeveler jobs, using one of the following methods:
  - Specifying the **bulkxfer** keyword in the LoadLeveler job command file.

    Example:

    ```
    #@ bulkxfer=yes
    ```

    If users specify this keyword for jobs that use the IP communication protocol, LoadLeveler ignores the **bulkxfer** keyword.
  - Specifying a POE line command parameter on interactive jobs.

    Example:

    ```
    poe_job -use_bulk_xfer=yes
    ```
  - Specifying an environment variable on interactive jobs.

    Example:

    ```
    export MP_USE_BULK_XFER=yes
        poe_job
    ```

    Note: Setting both the **MP_USE_BULK_XFER** environment variable *and* the **-use_bulk_xfer** POE command line option are *not* required. Set one or the other because the **-use_bulk_xfer** command line option will override the **MP_USE_BULK_XFER** environment variable.

Preparing a job for checkpoint/restart

You can checkpoint your entire job step, and allow a job step to restart from the last checkpoint.

LoadLeveler has the ability to checkpoint your entire job step, and to allow a job step to restart from the last checkpoint. When a job step is checkpointed, the entire state of each process of that job step is saved by the operating system. This checkpoint capability depends on MetaCluster HPC to perform checkpoint operations.

Use the information in Table 39 on page 181 to correctly configure your job for checkpointing.

Table 39. Checkpoint configurations

To specify that:	Do this:
Your job is checkpointable	• Add either one of the following two options to your job command file: 1. **checkpoint = yes** This enables your job to checkpoint in any of the following ways: – The application can initiate the checkpoint. – Checkpoint from a program which invokes the **ll_ckpt** API. – Checkpoint using the **llckpt** command. – As the result of a flush command. OR 2. **checkpoint = interval** This enables your job to checkpoint in any of the following ways: – The application can initiate the checkpoint. – Checkpoint from a program which invokes the **ll_ckpt** API. – Checkpoint using the **llckpt** command. – Checkpoint automatically taken by LoadLeveler. – As the result of a flush command. • If you would like your job to checkpoint itself, use **mpc_init_ckpt** for parallel jobs to cause the checkpoint to occur. • Only parallel jobs with a US adapter requirement can be checkpointed.
Your job step's executable is to be copied to the execute node	Add the **ckpt_execute_dir** keyword to the job command file.

Table 39. Checkpoint configurations (continued)

To specify that:	Do this:
LoadLeveler automatically checkpoints your job at preset intervals	1. Add the following option to your job command file: **checkpoint = interval** This enables your job to checkpoint in any of the following ways: • Checkpoint automatically at preset intervals • Checkpoint initiated from user application. • Checkpoint from a program which invokes the **ll_ckpt** API • Checkpoint using the **llckpt** command • As the result of a flush command 2. The system administrators must set the following two keywords in the configuration file to specify how often LoadLeveler should take a checkpoint of the job. These two keywords are: **MIN_CKPT_INTERVAL** = *number* Where *number* specifies the initial period, in seconds, between checkpoints taken for running jobs. **MAX_CKPT_INTERVAL** = *number* Where *number* specifies the maximum period, in seconds, between checkpoints taken for running jobs. The time between checkpoints will be increased after each checkpoint within these limits as follows: • The first checkpoint is taken after a period of time equal to the **MIN_CKPT_INTERVAL** has passed. • The second checkpoint is taken after LoadLeveler waits *twice as long* (**MIN_CKPT_INTERVAL** X 2) • The third checkpoint is taken after LoadLeveler waits twice as long again (**MIN_CKPT_INTERVAL** X 4) before taking the third checkpoint. LoadLeveler continues to double this period until the value of **MAX_CKPT_INTERVAL** has been reached, where it stays for the remainder of the job. A minimum value of 900 (15 minutes) and a maximum value of 7200 (2 hours) are the defaults. You can set these keyword values globally in the global configuration file so that all machines in the cluster have the same value, or you can specify a different value for each machine by modifying the local configuration files.
Your job will not be checkpointed	Add the following option to your job command file: • **checkpoint = no** This will disable checkpoint.

Table 39. Checkpoint configurations (continued)

To specify that:	Do this:
Your job has successfully checkpointed and terminated. The job has left the LoadLeveler job queue and you want LoadLeveler to restart your executable from an existing checkpoint file.	1. Add the following option to your job command file: • **restart_from_ckpt = yes** 2. When submitting a checkpointable job, specify the name of the checkpoint directory by setting the following job command file keywords to specify the directory and file name of the checkpoint directory to be used: • **ckpt_subdir** When the job command file is submitted, a new job will be started that uses the specified checkpoint directory to restart the previously checkpointed job. The job command file that was used to submit the original job should be used to restart from checkpoint. The only modifications to this file should be the addition of **restart_from_ckpt = yes** and ensuring **ckpt_subdir** points to the appropriate checkpoint directory.
Your job has successfully checkpointed. The job has been vacated and remains on the LoadLeveler job queue.	When the job restarts, if a checkpoint file is available, the job will be restarted from that file. If a checkpoint file is not available upon restart, the job will be started from the beginning.

Preparing a job for preemption

Depending on various configuration options, LoadLeveler may preempt your job so that a higher priority job step can run.

Administrators may:
- Configure LoadLeveler or external schedulers to preempt jobs through various methods.
- Specify preemption rules for job classes.
- Manually preempt your job using LoadLeveler interfaces.

To ensure that your job can be resumed after preemption, set the **restart** keyword in the job command file to **yes**.

Submitting a job command file

After building a job command file, you can submit it for processing either to a machine in the LoadLeveler cluster or one outside of the cluster.

See “Querying multiple LoadLeveler clusters” on page 74 for information on submitting a job to a machine outside the cluster. Use the **llsubmit** command to submit a job command file.

When you submit a job, LoadLeveler assigns a job identifier and one or more step identifiers.

The LoadLeveler job identifier consists of the following:

machine name
The name of the machine which assigned the job identifier.

jobid A number given to a group of job steps that were initiated from the same job command file.

The LoadLeveler step identifier consists of the following:

job identifier
The job identifier.

stepid A number that is unique for every job step in the job you submit.

If a job command file contains multiple job steps, every job step will have the same jobid and a unique stepid.

For an example of submitting a job, see Chapter 9, "Example: Using commands to build, submit, and manage jobs," on page 227.

In a multicluster environment, job and step identifiers are assigned by the local cluster and are retained by the job regardless of what cluster the job runs in.

Job state monitoring

A job can be submitted using the **llsubmit** command with the **-p** option, so that the submitter can be notified of when the state of the job step changes.

For example, issue:

```
llsubmit -p my_notification_program job1.cmd
```

where, *my_notification_program* is the full path name of the user notification program. The user notification program must be in a file system that is accessible by the job manager daemon managing the job. The owner of the job must have execute permission for the program.

Submitting a job using a submit-only machine

You can submit jobs from submit-only machines. Submit-only machines allow machines that do not run LoadLeveler daemons to submit jobs to the cluster. Use the **llsubmit** command to submit a job.

To install submit-only LoadLeveler, follow the procedure in the *LoadLeveler for AIX: Installation Guide* or *LoadLeveler for Linux: Installation Guide*.

In addition to allowing you to submit jobs, the submit-only feature allows you to cancel and query jobs from a submit-only machine.

Working with parallel jobs

LoadLeveler allows you to schedule parallel batch jobs that have been written using the following MPI implementations:

- On AIX and Linux:
  - IBM Parallel Environment Runtime Edition. Additional documentation is available from the IBM Clusters Information Center (http://publib.boulder.ibm.com/infocenter/clresctr/vxrx/index.jsp).
  - MPICH2 is an open-source, portable implementation of the Message-Passing Interface Standard (MPI-1 and MPI-2) developed by Argonne National

Laboratory. Additional documentation is available from the MPICH2 web site (http://www.mcs.anl.gov/research/projects/mpich2/).

  - Open MPI is an open source MPI-2 implementation that is developed and maintained by a consortium of academic, research, and industry partners. Additional documentation is available from the Open MPI web site (http://www.open-mpi.org/).

- On Linux:
  - Intel MPI is an MPI-2 implementation that focuses on making applications perform better on Intel architecture-based clusters. Additional documentation is available from the Intel MPI Library web site (http://software.intel.com/en-us/articles/intel-mpi-library/).

Step for controlling whether LoadLeveler copies environment variables to all executing nodes

You may specify that LoadLeveler is to copy, either to all executing nodes or to only the master executing node, the environment variables that are specified in the environment job command file statement for a parallel job.

Before you begin: You need to know:

- Whether Parallel Environment (PE) will be used to run the parallel job; if so, then LoadLeveler does not have to copy the application environment to the executing nodes.
- How to correctly specify the **env_copy** keyword. For information about the **env_copy** keyword syntax and other details, see "Job command file keyword descriptions" on page 335.
- To specify whether LoadLeveler is to copy environment variables to only the master node, or to all executing nodes, use the **#@ env_copy** keyword in the job command file.

Ensuring that parallel jobs in a cluster run on the correct levels of PE and LoadLeveler software

If support for parallel POE jobs is required, users must be aware that when LoadLeveler uses Parallel Environment for parallel job submission, that the PE software requires the same level of PE to be used throughout the parallel job. Different levels of PE cannot be mixed.

In addition, the level of PE dependencies, including LAPI, also need to be consistent. For instance, different levels of LAPI are required on AIX 5.3 and AIX 6.1. The **requirements** keyword of the job command file can be used to ensure that all the tasks of a POE job run on compatible levels of PE, PE dependencies, and LoadLeveler software in a cluster. For example, use of a requirements statement similar to one of the two following statements to select machines running the same levels of AIX, will ensure that a parallel job will run on nodes where consistent levels are installed.

- # @ requirements = (OpSys == "AIX61")
- # @ requirements = (OpSys == "AIX71")

If a statement such as `# @ executable = /bin/poe` is specified in a job command file, and if the job is intended to be run on machines with PE 1.1 installed, then it is important that the job be submitted from a machine with PE 1.1 installed. When the "executable" keyword is used, LoadLeveler will copy the associated binary on the submitting machine and send it to a running machine for execution. In this

example, the POE program will fail if the submitting and the running machines are at different PE software levels. This problem can be circumvented by not using the executable keyword in the job command file. By omitting this keyword, the job command file itself is the shell script that will be executed. If this script invokes a local version of the POE binary then there is no compatibility problem at run time.

Task-assignment considerations

You can use the keywords listed in Table 40 to specify how LoadLeveler assigns tasks to nodes. With the exception of unlimited blocking, each of these methods prioritizes machines in an order based on their **MACHPRIO** expressions. Various task assignment keywords can be used in combination, and others are mutually exclusive.

Table 40. Valid combinations of task assignment keywords are listed in each column

Keyword	Valid Combinations				
total_tasks	X	X			
tasks_per_node			X	X	
node = <min, max>			X		
node = <number>	X			X	
task_geometry					X
blocking		X			

The following examples show how each allocation method works. For each example, consider a 3-node cluster with machines named "N1," "N2," and "N3". The machines' order of priority, according to the values of their MACHPRIO expressions, is: N1, N2, N3. N1 has 4 initiators available, N2 has 6, and N3 has 8.

node and total_tasks

When you specify the **node** keyword with the **total_tasks** keyword, the assignment function will allocate all of the tasks in the job step evenly among however many nodes you have specified. If the number of **total_tasks** is not evenly divisible by the number of nodes, then the assignment function will assign any larger groups to the first nodes on the list that can accept them. In this example, 14 tasks must be allocated among 3 nodes:

```
# @ node=3
# @ total_tasks=14
```

Table 41 shows the machine, available initiators, and assigned tasks:

Table 41. node and total_tasks

Machine	Available Initiators	Assigned Tasks
N1	4	4
N2	6	5
N3	8	5

The assignment function divides the 14 tasks into groups of 5, 5, and 4, and begins at the top of the list, to assign the first group of 5. The assignment function starts at N1, but because there are only 4 available initiators, cannot assign a block of 5 tasks. Instead, the function moves down the list and assigns the two groups of 5 to N2 and N3, the assignment function then goes back and assigns the group of 4 tasks to N1.

node and tasks_per_node

When you specify the node keyword with the **tasks_per_node** keyword, the assignment function will assign tasks in groups of the specified value among the specified number of nodes.

```
# @ node = 3
# @ tasks_per_node = 4
```

node_topology and island_count

Nodes can be selected based on the islands they belong to by specifying **node_topology=island**. When **node_topology** is used, the number of nodes can only be specified using the **node** keyword.

The number of nodes requested by the node keyword are selected from among **island_count** islands. For example:

```
# @ node = 32
# @ tasks_per_node = 16
# @ node_topology = island
# @ island_count = 1
```

All 32 nodes for the job step are from the same island. However, in this example:

```
# @ node = 32
# @ tasks_per_node = 16
# @ node_topology = island
# @ island_count = 2
```

The 32 nodes are selected from among 2 islands.

For more information about the **node_topology** and **island_count** keywords, see "Job command file keyword descriptions" on page 335.

blocking

When you specify blocking, tasks are allocated to machines in groups (blocks) of the specified number (blocking factor). The assignment function will assign one block at a time to the machine which is next in the order of priority until all of the tasks have been assigned. If the total number of tasks are not evenly divisible by the blocking factor, the remainder of tasks are allocated to a single node. The blocking keyword must be specified with the total_tasks keyword. For example:

```
# @ blocking = 4
# @ total_tasks = 17
```

Where **blocking** specifies that a job's tasks will be assigned in blocks, and **4** designates the size of the blocks. Table 42 shows how a blocking factor of 4 would work with 17 tasks:

Table 42. Blocking

Machine	Available Initiators	Assigned Tasks
N1	4	4
N2	6	5
N3	8	8

The assignment function first determines that there will be 4 blocks of 4 tasks, with a remainder of one task. Therefore, the function will allocate the remainder with the first block that it can. N1 gets a block of four tasks, N2 gets a block, plus the remainder, then N3 gets a block. The assignment function begins again at the top

of the priority list, and N3 is the only node with enough initiators available, so N3 ends up with the last block.

unlimited blocking

When you specify unlimited blocking, the assignment function will allocate as many jobs as possible to each node; the function prioritizes nodes primarily by how many initiators each node has available, and secondarily on their MACHPRIO expressions. This method allows you to allocate tasks among as few nodes as possible. To specify unlimited blocking, specify "unlimited" as the value for the blocking keyword. The **total_tasks** keyword must also be specified with unlimited blocking. For example:

```
# @ blocking = unlimited
# @ total_tasks = 17
```

Table 43 lists the machine, available initiators, and assigned tasks for unlimited blocking:

Table 43. Unlimited blocking

Machine	Available Initiators	Assigned Tasks
N3	8	8
N2	6	6
N1	4	3

The assignment function begins with N3 (because N3 has the most initiators available), and assigns 8 tasks, N2 takes six, and N1 takes the remaining 3.

task_geometry

The **task_geometry** keyword allows you to specify which tasks run together on the same machines, although you cannot specify which machines. In this example, the **task_geometry** keyword groups 7 tasks to run on 3 nodes:

```
# @ task_geometry = {(5,2)(1,3)(4,6,0)}
```

The entire **task_geometry** expression must be enclosed within braces. The task IDs for each node must be enclosed within parenthesis, and must be separated by commas. The entire range of task IDs that you specify must begin with zero, and must end with the task ID which is one less than the total number of tasks. You can specify the task IDs in any order, but you cannot skip numbers (the range of task IDs must be complete). Commas may only appear between task IDs, and spaces may only appear between nodes and task IDs.

Submitting jobs that use striping

When communication between parallel tasks occurs only over a single device such as en0, the application and the device are gated by each other. The device must wait for the application to fill a communication buffer before it transmits the buffer and the application must wait for the device to transmit and empty the buffer before it can refill the buffer. Thus the application and the device must wait for each other and this wastes time.

The technique of striping refers to using two or more communication paths to implement a single communication path as perceived by the application. As the application sends data, it fills up a buffer on one device. As that buffer is transmitted over the first device, the application's data begins filling up a second buffer and the application perceives no delay in being able to write. When the second buffer is full, it begins transmission over the second device and the application moves on to the next device. When all devices have been used, the

application returns to the first device. Much, if not all of the buffer on the first device has been transmitted while the application wrote to the buffers on the other devices so the application waits for a minimal amount of time or possibly does not wait at all.

LoadLeveler supports striping in two ways. When multiple switch planes or networks are present, striping over them is indicated by requesting **sn_all** (multiple networks).

If multiple adapters are present on the same network and the communication subsystem, such as LAPI, supports striping over multiple adapters on the same network, specifying the **instances** keyword on the network statement requests striping over adapters on the same network. The **instances** keyword specifies the number of adapters on a single network to stripe on. It is possible to stripe over multiple networks and over multiple adapters on each network by specifying both **sn_all** and a value for **instances** greater than one.

- **User space striping:** When **sn_all** is specified on a network statement with **US** mode, LoadLeveler commits an equivalent set of adapter resources (adapter windows and memory) on each of the networks present in the system to the job on each node where the job runs. The communication subsystem is initialized to indicate that it should use the user space communication protocol on all the available switch adapters to service communication requests on behalf of the application.
- **IP striping:** When the **sn_all** device is specified on a network statement with the **IP** mode, LoadLeveler attempts to locate the striped IP address associated with the switch adapters, known as the multi-link address. If it is successful, it passes the multi-link address to POE for use. If multi-link addresses are not available, LoadLeveler instructs POE to use the IP address of one of the switch adapters. The IP address that is used is different each time a choice has to be made in an attempt to balance the adapter use. Multi-link addresses must be configured on the system prior to running LoadLeveler. If a multi-link address is specified for a node, LoadLeveler assigns the multi-link address and multi-link IP name to the striping adapter on that node. If a multi-link address is not present on a node, the **sn_all** adapter associated with the node will not have an IP address or IP name. If not all of the nodes of a system have multi-link addresses but some do, LoadLeveler will only dispatch jobs that request IP striping to nodes that have multi-link addresses.

  Jobs that request striping (both user space and IP) can be submitted to nodes with only one switch adapter. In that situation, the result is the same as if the job requested no striping.

  Note: When configured, a multi-link address is associated with the virtual **ml0** device. The IP address of this device is the multi-link address. As with any other device with an IP address, the **ml0** device can be requested in IP mode on the network statement. Doing so would yield a comparable effect to requesting **sn_all** IP except that no checking would be performed by LoadLeveler to ensure the associated adapters are actually working. Thus it would be possible to dispatch a job that requested communication over **ml0** only to have the job fail because the switch adapters that **ml0** stripes over were down.
- **Striping over one network:** If the **instances** keyword is specified on a network statement with a value greater than one, LoadLeveler allocates multiple sets of resources for the protocol using as many sets as the **instances** keyword specified. For User Space jobs, these sets are adapter windows and memory. For IP jobs, these sets are IP addresses. If multiple adapters exist on each node on the same network, then these sets of adapter resources will be distributed among

all the available adapters on the same network. Even though LoadLeveler will allocate resources to support striping over a single network, the communication subsystem must be capable of exploiting these resources in order for them to be used.

Understanding striping over multiple networks

Striping over multiple networks involves establishing a communication path using one or more of the available communication networks or switch fabrics. How those paths are established depends on the network adapter that is present. For the SP Switch2 family of adapters, it is not necessary to acquire communication paths among all tasks on all fabrics as long as there is at least one fabric over which all tasks can communicate. However, each adapter on a machine, if it is available, must use exactly the same adapter resources (window and memory amount) as the other adapters on that machine. Switch adapters are not required to use exactly the same resources on each network, but in order for a machine to be selected, there must be an available communication path on all networks.

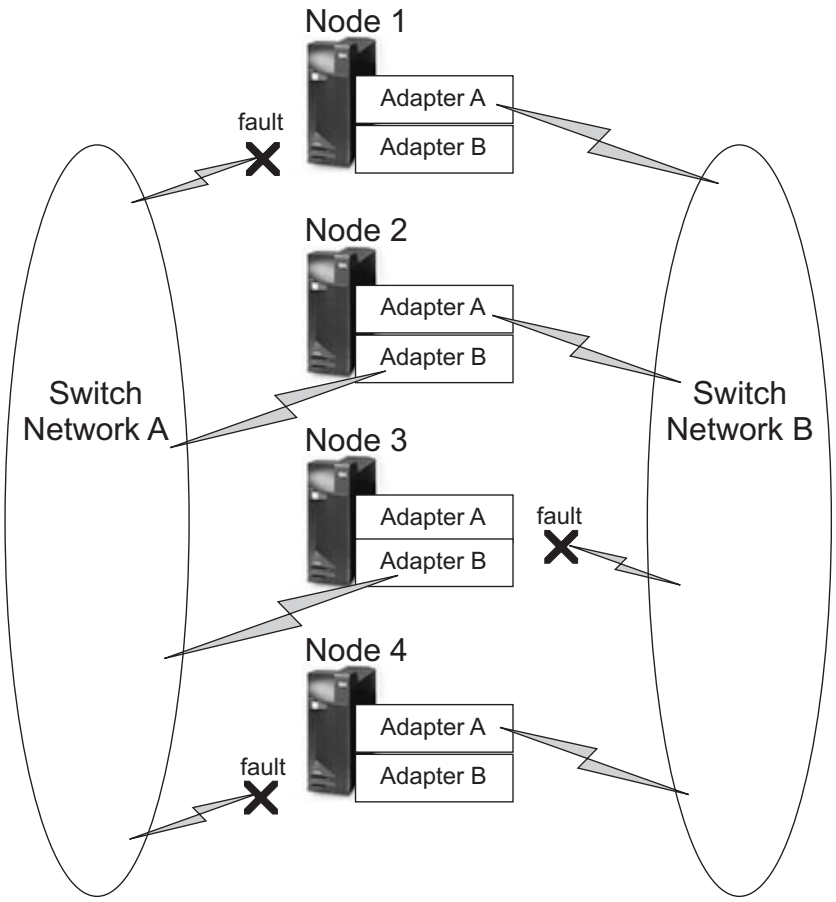

Figure 17. Striping over multiple networks

Consider these sample scenarios using the network configuration as shown in Figure 17 where the adapters are from the SP Switch2 family:

- If a three node job requests striping over networks, it will be dispatched to Node 1, Node 2 and Node 4 where it can communicate on Network B as long as the adapters on each machine have a common window free and sufficient memory available. It cannot run on Node 3 because that node only has a common communication path with Node 2, namely Network A.

- If a three node job does not request striping, it will not be run because there are not enough adapters connected to Network A to run the job. Notice both the adapter connected to Network A on Node 1 and the adapter connected to Network A on Node 4 are both at fault. SP Switch2 family adapters can only use the adapter connected to Network A for non-striped communication.
- If a three node job requests striped IP and some but not all of the nodes have multi-linked addresses, the job will only be dispatched to the nodes that have the multi-link addresses.

Consider these sample scenarios using the network configuration as shown in Figure 17 on page 190 where the adapters are switch adapters:

- If a three node job requests striping over networks, it will not be dispatched because there are not three nodes that have active connections to both networks.
- If a three node job does not request striping, it can be run on Node 1, Node 2, and Node 4 because they have an active connection to network B.
- If a three node job requests striped IP and some but not all of the nodes have multi-linked addresses, the job will only be dispatched to the nodes that have the multi-link addresses.

Note that for all adapter types, adapters are allocated to a step that requests striping based on what the node knows is the available set of networks or fabrics. LoadLeveler expects each node to have the same knowledge about available networks. If this is not true, it is possible for tasks of a step to be assigned adapters which cannot communicate with tasks on other nodes.

Similarly, LoadLeveler expects all adapters that are identified as being on the same Network ID or fabric ID to be able to communicate with each other. If this is not true, such as when LoadLeveler operates with multiple, independent sets of networks, other attributes of the Step, such as the requirements expression, must be used to ensure that only nodes from a single network set are considered for the step.

As you can see from these scenarios, LoadLeveler will find enough nodes on the same communication path to run the job. If enough nodes connected to a common communication path cannot be found, no communication can take place and the job will not run.

Understanding striping over a single network

Striping over a single network is only supported by switch adapters.

Figure 18 on page 192 shows a network configuration where the adapters support striping over a single network.

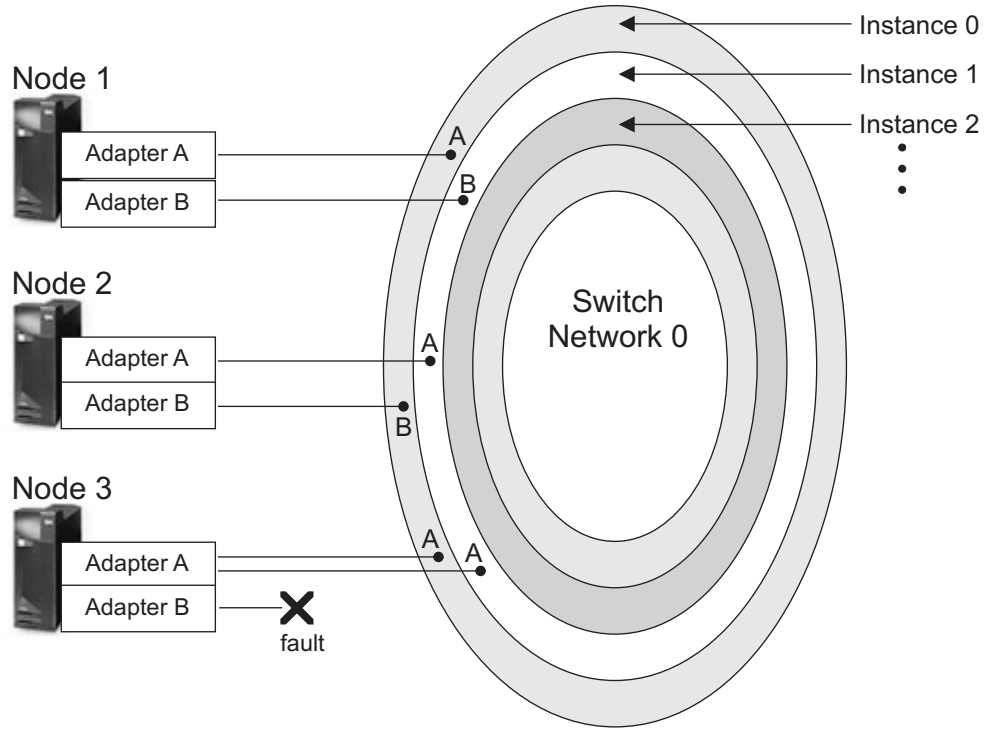

Figure 18. Striping over a single network

Both Adapter A and Adapter B on a node are connected to Network 0. The entire oval represents the physical network and the concentric ovals (shaded differently) represent the separate communication paths created for a job by the **instances** keyword on the network statement. In this case a three node job requests two instances for communication. On Node 1, adapter A is used for instance 0 and adapter B is used for instance 1. There is no requirement to use the same adapter for the same instance so on Node 2, adapter B was used for instance 0 and adapter A for instance 1.

On Node 3, where a fault is keeping adapter B from connecting to the network, adapter A is used for both instance 0 and instance 1 and Node 3 is available for the job to use.

The network itself does not impose any limitation on the total number of communication paths that can be active at a given time for either a single job or all the jobs using the network. As long as nodes with adapter resources are available, additional communication paths can be created.

Examples: Requesting striping in network statements

You request that a job be run using striping with the **network** statement in your job command file. The default when instances is not specified for a job in the **network** statement is controlled by the class stanza keyword for **sn_all**. For more information on the **network** and **max_protocol_instances** statements, see the keyword descriptions in "Job command file keyword descriptions" on page 335.

Shown here are examples of IP and user space network modes:

- **Example 1: Requesting striping using IP mode**

  To submit a job using IP striping, your network statement would look like this:

  ```
  network.MPI = sn_all,,IP
  ```

- **Example 2: Requesting striping using user space mode**

  To submit a job using user space striping, your network statement would look like this:

  ```
  network.MPI = sn_all,,US
  ```

- **Example 3: Requesting striping over a single network**

  To request IP striping over multiple adapter on a single network, the network statement would look like this:

  ```
  network.MPI = sn_single,,IP,,instances=2
  ```

  If the nodes on which the job runs have two or more adapters on the same network, two different IP addresses will be allocated to each task for MPI communication. If only one adapter exists per network, the same IP address will be used twice for each task for MPI communication.

- **Example 4: Requesting striping over multiple networks and multiple adapters on the same network**

  To submit a user space job that will stripe MPI communication over multiple adapters on all networks present in the system the network statement would look like this:

  ```
  network.MPI = sn_all,,US,,instances=2
  ```

  If, on a node where the job runs, there are two adapters on each of the two networks, one adapter window would be allocated from each adapter for MPI communication by the job. If only one network were present with two adapters, one adapter window from each of the two adapters would be used. If two networks were present but each only had one adapter on it, two adapter windows from each adapter would be used to satisfy the request for two instances.

Running interactive POE jobs

POE will accept LoadLeveler job command files. However, you can still set the following environment variables to define specific LoadLeveler job attributes before running an interactive POE job:

LOADL_ACCOUNT_NO
: The account number associated with the job.

LOADL_INTERACTIVE_CLASS
: The class to which the job is assigned.

MP_TASK_AFFINITY
: The affinity preferences requested for the job.

For information on other POE environment variables, see *IBM Parallel Environment Runtime Edition for AIX Operation and Use*, SC23-6781.

For an interactive POE job, LoadLeveler does not start the POE process therefore LoadLeveler has no control over the process environment or resource limits.

You also may run interactive POE jobs under a reservation. For additional details about reservations and submitting jobs to run under them, see "Working with reservations" on page 203.

Interactive POE jobs cannot be submitted to a remote cluster.

Debugging interfaces between POE and LoadLeveler

POE interacts with LoadLeveler by calling a set of interfaces between POE and LoadLeveler, which are provided in the LoadLeveler product. The LL_POE_DEBUG environment variable can be used to trace the interaction from the LoadLeveler side, mainly for debugging purposes.

When LL_POE_DEBUG is set to `yes`, the interfaces between POE and LoadLeveler (called by POE processes) write LoadLeveler debug messages to `standard error` (the screen or a file, whichever was specified for other `standard error` output from the POE process).

When LL_POE_DEBUG is set to a file path name, the interfaces between POE and LoadLeveler (called by POE and PMD processes) write LoadLeveler debug messages to the following trace files (whose names include the specified file path and the process ID):

POE process trace file
file_path_poe_process_id.poe_trace

PMD process trace file
file_path_pmd_process_id.pmd_trace

Note: When you set LL_POE_DEBUG to a file path name, the specified file path does not have to be a directory; it is simply a prefix in the trace file's name.

If one or both of the trace files cannot be created due to a file access error, the job itself continues and is not affected.

The following example shows the output when the trace files can be created:

```
$ LL_POE_DEBUG=/tmp/myjob poe date -hfile /tmp/h -resd yes
Wed Jun  1 14:36:05 EDT 2011

$ ls /tmp/myjob*
/tmp/myjob_8708.poe_trace /tmp/myjob_8738.pmd_trace
```

The following example shows the output when the trace files go to standard error:

```
$ LL_POE_DEBUG=yes poe date -hfile /tmp/h -resd yes
06/01 14:34:49.355270 pe_rm_init: rmapi_version=1300 from caller and 1300 from LoadLeveler
06/01 14:34:49.355317 pe_rm_init: listen_socket=3, ll_get_data returns 0
06/01 14:34:49.355379 pe_rm_submit_job: job_format=JOB_OBJECT
06/01 14:34:49.355388 pe_rm_submit_job: num_nodes=-1
06/01 14:34:49.355473 host 0: c197blade4b10.ppd.pok.ibm.com
....
06/01 14:34:50.521802 pe_rm_get_event: JOB_TIMER_EVENT: no event data.
Wed Jun  1 14:34:50 EDT 2011
06/01 14:34:52.664209 pe_rm_free: start
06/01 14:34:53.745940 pe_rm_free: ll_close was called.
06/01 14:34:53.746244 pe_rm_free: ll_deallocate_job was called.
06/01 14:34:53.746260 pe_rm_free: return
```

Running MPICH2

LoadLeveler for AIX and LoadLeveler for Linux support the MPICH2 open-source implementation of the Message-Passing Interface (MPI).

MPICH2 is an open-source, portable implementation of the MPI Standard developed by Argonne National Laboratory. It contains a complete implementation of version 2 of the MPI Standard and also significant parts of MPI-2, particularly in the area of parallel I/O. MPICH2 MPI implementation is supported by

LoadLeveler for AIX and LoadLeveler for Linux. Additional documentation is available from the MPICH2 web site (http://www.mcs.anl.gov/research/projects/mpich2/).

To run MPICH2 jobs in LoadLeveler, prepare a job command file and specify the **MPICH** job type. For launching MPI jobs, specify an MPICH2 provided executable in the executable statement or in the body of the job command file (when no executable statement is used). For **MPICH** job types, LoadLeveler allocates the machines to run the parallel job and starts the job command file or the specified executable as the master task.

To designate the number of tasks to start and where to start them, specify the LoadLeveler **LOADL_TOTAL_TASKS** and **LOADL_HOSTFILE** run-time environment variables in the job command file as arguments to the MPICH2 executable. Also, specify the LoadLeveler **llspawn.stdio** executable as the remote command for the MPICH2 executable to use when launching MPI tasks.

The following standard **mpirun** script options are not supported:

- ckpoint-interval
- ckpoint-prefix
- ckpoint-num
- ckpointlib

Checkpoint in MPICH2 is supported by the BLCR library, which cannot be embedded into the framework of LoadLeveler, so the checkpoint related options are not supported by LoadLeveler.

Sample programs are available:

- See "MPICH2 sample job command file" on page 198 for a sample MPICH2 job command file.
- The LoadLeveler samples directory also contains sample files:
  - On AIX, use directory `/usr/lpp/LoadL/full/samples/llmpich`
  - On Linux, use directory `/opt/ibmll/LoadL/full/samples/llmpich`

  These sample files include:
  - `ivp.c`: A simple MPI application that you may run as an MPICH2 job.
  - Job command files to run the `ivp.c` program as a batch job:

    For MPICH2: `mpich_ivp.cmd`

Running Open MPI

To run Open MPI jobs in LoadLeveler, do the following:

1. Prepare a job command file and specify the MPICH job type. For MPICH job types, LoadLeveler allocates the machines to run the parallel job and starts the job command file or the specified executable as the master task.
2. Specify an Open MPI provided executable for launching MPI jobs in the body of the job command file, for example **mpirun**.

Notes:

1. When running with Open MPI 1.5.4, you do not have to specify any special options to run the Open MPI job with LoadLeveler.
2. When running versions of Open MPI prior to 1.5.4
   a. You must specify the LoadLeveler **LOADL_TOTAL_TASKS** and **LOADL_HOSTFILE** run-time environment variables as arguments to the

Open MPI executable to specify the number of tasks to start and where to start them. For more information about these run-time environment variables, see "Run-time environment variables" on page 384.

b. You must also specify the LoadLeveler **llspawn.stdio** executable as the remote command for the Open MPI executable to use when launching MPI tasks. In addition, specify the **--leave-session-attached** option, which will keep the spawned MPI tasks descendents of LoadLeveler processes. This is a necessary step for LoadLeveler jobs.

For examples that show how to run Open MPI jobs, see:

- "Open MPI 1.5.4 sample job command file" on page 200
- The LoadLeveler samples directory (**LoadL/full/samples/llmpich**), which includes:
  - **ivp.c**: a simple MPI application that you can run as an Open MPI job
  - Job command files to run the **ivp.c** program as a batch job. The job command file to use for Open MPI is: **open_mpirun_ivp.cmd**.

Running Intel MPI jobs

To run Intel MPI jobs in LoadLeveler, follow these steps:

1. Prepare a job command file and specify MPICH as the job type.
2. In the body of the job command file, specify an Intel MPI-provided executable for launching MPI jobs, for example, **mpiexec.hydra**.

   Notes:

   a. The LoadLeveler **#@executable** directive cannot be used to specify the executable provided by Intel MPI.

   b. For MPICH job types, LoadLeveler allocates the machines to run the parallel job and starts the job command file or the specified executable as the master task.
3. When running with Intel MPI 4.0.3, specify the LoadLeveler **llspawn.stdio** executable as the remote command for the Intel MPI executable to use when launching MPI tasks.
4. When running with Intel MPI 4.0.2, define the number of tasks (**-n**) to start and where to start them (**-f**) by specifying the LoadLeveler **LOADL_TOTAL_TASKS** and **LOADL_HOSTFILE** run-time environment variables in the job command file as arguments to the Intel MPI executable. See "Run-time environment variables" on page 384 for more information.

For examples that show how to run Intel MPI jobs, see:

- "Intel MPI 4.0.3 sample job command file" on page 199 and "Intel MPI 4.0.2 sample job command file" on page 199
- The LoadLeveler samples directory (**LoadL/full/samples/llmpich**), which includes:
  - **ivp.c**: a simple MPI application that you can run as an Intel MPI job.
  - Job command files to run the **ivp.c** program as a batch job. The job command file to use for Intel MPI is: **intel_hydra_ivp.cmd**.

Running embarassingly parallel jobs

Located in the **LoadL/resmgr/full/samples/autonomous** subdirectory, sample korn shell scripts are provided, which show how an embarrassingly parallel job can be

run as a LoadLeveler job. The sample shell scripts can be used as is or can be modified to provide a more installation-specific solution for running embarrassingly parallel jobs. Two sample shell scripts are provided. The first is **autonomous_master.ksh** which runs on the master node and uses **llspawn.stdio** to start a second script, **autonomous_slave.ksh** on each allocated node. **autonomous_slave.ksh** starts the tasks designated to run on a node. The two scripts are designed to run the embarrassingly parallel job in one of two ways.

- A specified command will be invoked in each initiator (or task) allocated for the job.
- Commands from a specified command file are invoked in each initiator (or task) allocated for the job.

More commands can be specified than there are initiators allocated for the job. If so, the scripts will start as many commands as there are initiators, then wait for commands to complete, and start subsequent commands until all commands have run.

Examples: Building parallel job command files

This topic contains sample job command files for the following parallel environments:

- IBM AIX Parallel Operating Environment (POE)
- MPICH2
- Intel Message Passing Interface (MPI) 4.0.3 and 4.0.2
- Open MPI 1.5.4
- Embarassingly parallel

POE sample job command file

This is a sample job command file for POE:

```
#
# @ job_type = parallel
# @ environment = COPY_ALL
# @ output = poe.out
# @ error = poe.error
# @ node = 8,10
# @ tasks_per_node = 2
# @ network.LAPI = sn_all,US,,instances=1
# @ network.MPI = sn_all,US,,instances=1
# @ wall_clock_limit = 60
# @ executable = /usr/bin/poe
# @ arguments = /u/richc/My_POE_program -euilib "us"
# @ class = POE
# @ queue
```

In this example:

- The total number of nodes requested is a minimum of eight and a maximum of 10 (**node=8,10**). Two tasks run on each node (**tasks_per_node=2**). Thus the total number of tasks can range from 16 to 20.
- Each task of the job will run using the LAPI protocol in US mode with a switch adapter (**network.LAPI=sn_all,,US,instances=1**), and using the MPI protocol in US mode with a switch adapter (**network.MPI=sn_all,,US,instances=1**).
- The maximum run time allowed for the job is 60 seconds (**wall_clock_limit=60**).

This is another sample job command file for POE:

```
#
# @ job_type = parallel
# @ input = poe.in.1
# @ output = poe.out.1
```

```
# @ error = poe.err
# @ node = 2,8
# @ network.MPI = sn_single,shared,IP
# @ wall_clock_limit = 60
# @ class = POE
# @ queue
/usr/bin/poe /u/richc/my_POE_setup_program -infolevel 2
/usr/bin/poe /u/richc/my_POE_main_program -infolevel 2
```

In this example:

- POE is invoked twice, through **my_POE_setup_program** and **my_POE_main_program**.
- The job requests a minimum of two nodes and a maximum of eight nodes (**node=2,8**).
- The job by default runs one task per node.
- The job uses the MPI protocol with a switch adapter in IP mode (**network.MPI=sn_single,shared,IP**).
- The maximum run time allowed for the job is 60 seconds (**wall_clock_limit=60**).

MPICH2 sample job command file

This is a sample job command file for MPICH2:

```
# ! /bin/ksh
# LoadLeveler JCF file for running an MPICH2 job
# @ job_type = MPICH
# @ node = 4
# @ tasks_per_node = 2
# @ output = mpich2_test.$(cluster).$(process).out
# @ error = mpich2_test.$(cluster).$(process).err
# @ queue
echo "----------------------------------------------------------"
echo LOADL_STEP_ID=$LOADL_STEP_ID
echo "----------------------------------------------------------"

/opt/mpich2/bin/mpirun -n $LOADL_TOTAL_TASKS -f \
$LOADL_HOSTFILE -launcher-exec /opt/ibmll/LoadL/full/bin/llspawn.stdio \
/common/NFS/ll_bin/mpich2_test
```

In the body of the job command file, the **mpirun** command is invoked. The **mpirun** arguments specified are:

-n Specifies the number of parallel processes.

LOADL_TOTAL_TASKS
Is the environment variable set by LoadLeveler with the number of parallel processes of the job step.

-f Specifies the machine list file.

LOADL_HOSTFILE
Is the environment variable set by LoadLeveler with the file name that contains host names assigned to the parallel job step.

-launcher-exec
Specifies the launcher program used in MPICH2. **/opt/ibmll/LoadL/full/bin/llspawn.stdio** is the path to the LoadLeveler provided remote command for launching MPI tasks.

The following is another example of a MPICH2 job command file:

```
# ! /bin/ksh
# LoadLeveler JCF file for running an MPICH2 job
# @ job_type = MPICH
# @ node = 4
# @ tasks_per_node = 2
```

```
# @ output = mpich2_test.$(cluster).$(process).out
# @ error = mpich2_test.$(cluster).$(process).err
# @ executable = /opt/mpich2/bin/mpirun
# @ arguments = -n $LOADL_TOTAL_TASKS -f $LOADL_HOSTFILE \
-launcher-exec /opt/ibmll/LoadL/full/bin/llspawn.stdio \
# @ queue
```

In this example:

- The **mpirun** script is specified as a value of the executable job command file keyword.
- The following **mpirun** script arguments are specified with the arguments job command file keyword:

  ```
  -n $LOADL_TOTAL_TASKS -f $LOADL_HOSTFILE -launcher-exec
   /opt/ibmll/LoadL/full/bin/llspawn.stdio/common/NFS/ll_bin/mpich2_test
  ```

  -n
  Specifies the number of parallel processes.

  LOADL_TOTAL_TASKS
  Is the environment variable set by LoadLeveler with the number of parallel processes of the job step.

  -f Specifies the machine list file.

  LOADL_HOSTFILE
  Is the environment variable set by LoadLeveler with file name, which contains host names assigned to the parallel job step.

  -launcher-exec
  Specifies the launcher program used in MPICH2. **/opt/ibmll/LoadL/full/bin/llspawn.stdio** is the path to the LoadLeveler provided remote command for launching MPI tasks.

Intel MPI 4.0.3 sample job command file

This is a sample job command file for Intel MPI 4.0.3:

```
# ! /bin/ksh
# LoadLeveler JCF file for running an Intel MPI job
# @ job_type = MPICH
# @ node = 4
# @ tasks_per_node = 2
# @ output = intel_hydra_test.$(cluster).$(process).out
# @ error = intel_hydra_test.$(cluster).$(process).err
# @ queue
mpiexec.hydra -bootstrap ll \
/u/dave/mpi/intel_mpi/test_bin/excersize00A
```

Intel MPI 4.0.2 sample job command file

This is a sample job command file for Intel MPI 4.0.2:

```
# ! /bin/ksh
# LoadLeveler JCF file for running an Intel MPI job
# @ job_type = MPICH
# @ node = 4
# @ tasks_per_node = 2
# @ output = intel_hydra_test.$(cluster).$(process).out
# @ error = intel_hydra_test.$(cluster).$(process).err
# @ queue
mpiexec.hydra -n $LOADL_TOTAL_TASKS \
-f $LOADL_HOSTFILE \
-bootstrap ssh \
-bootstrap-exec llspawn.stdio \
/u/dave/mpi/intel_mpi/test_bin/excersize00A
```

In the body of the previous Intel MPI job command files, the **mpiexec.hydra** command is invoked. The specified **mpiexec.hydra** arguments are:

-n
Specifies the number of parallel processes.

LOADL_TOTAL_TASKS
Is the environment variable set byLoadLeveler with the number of parallel processes of the job step.

-f
Specifies the file of machine names on which to run MPI tasks.

LOADL_HOSTFILE
Is the environment variable set by LoadLeveler with the file name of the file that contains host names assigned to the parallel job step.

-bootstrap
Specifies that the bootstrap server protocol used by **mpiexec.hydra**. **rsh** is specified.

-bootstrap-exec
Specifies the executable to be used as the bootstrap server by **mpiexec.hydra**. **/opt/ibmll/LoadL/full/bin/llspawn.stdio** is the path to the LoadLeveler provided remote command for launching MPI tasks.

Open MPI 1.5.4 sample job command file

This is a sample job command file for Open MPI 1.5.4:

```
# ! /bin/ksh
# LoadLeveler JCF file for running an Open MPI job
# @ job_type = MPICH
# @ node = 4
# @ tasks_per_node = 2
# @ output = ompi_test.$(cluster).$(process).out
# @ error = ompi_test.$(cluster).$(process).err
# @ queue
mpirun /hpcc/dave/mpi/testdir/test_bin/excersize00A
```

In this sample Open MPI job command file, the **mpirun** command is invoked.

No additional **mpirun** options are necessary to run the Open MPI job with LoadLeveler.

Embarrassingly parallel sample job command file

The following job command file, **onecmd.cmd**, demonstrates how to run an embarrassingly parallel job using the two shell scripts from the **LoadL/resmgr/full/samples/autonomous** subdirectory:

```
#! /bin/ksh
# @ job_type=MPICH
# @ output = /hpcc/dave/mpi/testdir/embarrass.$(host).$(jobid).$(stepid).out
# @ error = /hpcc/dave/mpi/testdir/embarrass.$(host).$(jobid).$(stepid).err
# @ class = No_Class
# @ node = 3
# @ total_tasks = 6
# @ queue
/hpcc/dave/embarrass/autonomous_master.ksh -c /hpcc/dave/embarrass/onecmd.ksh -n 6
```

In this case the command specified as an argument to **autonomous_master.ksh** using **-c** is a shell script containing:

```
#!/bin/ksh
echo "stdout: Hello world from `hostname` task $LL_SAMPLE_TASK_ID"
echo "stderr: Hello world from `hostname` task $LL_SAMPLE_TASK_ID" >&2
```

When the job finished running, the file of allocated machines (from the **LOADL_HOSTFILE** environment variable) contained:

```
c111bc4n01.ppd.pok.ibm.com
c111bc4n01.ppd.pok.ibm.com
c111bc4n02.ppd.pok.ibm.com
c111bc4n02.ppd.pok.ibm.com
c111bc4n03.ppd.pok.ibm.com
c111bc4n03.ppd.pok.ibm.com
```

The following files were produced by the job:

```
embarrass.c111bc4n01.433.0.out.0
stdout: Hello world from c111bc4n01 task 0
embarrass.c111bc4n01.433.0.err.0
stderr: Hello world from c111bc4n01 task 0
embarrass.c111bc4n01.433.0.out.1
stdout: Hello world from c111bc4n01 task 1
embarrass.c111bc4n01.433.0.err.1
stderr: Hello world from c111bc4n01 task 1
embarrass.c111bc4n01.433.0.out.2
stdout: Hello world from c111bc4n02 task 2
embarrass.c111bc4n01.433.0.err.2
stderr: Hello world from c111bc4n02 task 2
embarrass.c111bc4n01.433.0.out.3
stdout: Hello world from c111bc4n02 task 3
embarrass.c111bc4n01.433.0.err.3
stderr: Hello world from c111bc4n02 task 3
embarrass.c111bc4n01.433.0.out.4
stdout: Hello world from c111bc4n03 task 4
embarrass.c111bc4n01.433.0.err.4
stderr: Hello world from c111bc4n03 task 4
embarrass.c111bc4n01.433.0.out.5
stdout: Hello world from c111bc4n03 task 5
embarrass.c111bc4n01.433.0.err.5
stderr: Hello world from c111bc4n03 task 5
```

Obtaining status of parallel jobs

Both end users and LoadLeveler administrators can obtain status of parallel jobs in the same way as they obtain status of serial jobs by using the **llq** command. By issuing **llq -l**, users get a list of machines allocated to the parallel job. If you also need to see task instance information use the **-x** option in addition to the **-l** option (**llq -l -x**). See *LoadLeveler: Command and API Reference* for samples of output using the **-x** and **-l** options with the **llq** command.

Obtaining allocated host names

llq -l output includes information on allocated host names. Another way to obtain the allocated host names is with the LOADL_PROCESSOR_LIST environment variable, which you can use from a shell script in your job command file as shown in the following example.

This example uses LOADL_PROCESSOR_LIST to perform a remote copy of a local file to all of the nodes, and then invokes POE. Note that the processor list contains an entry for each task running on a node. If two tasks are running on a node, LOADL_PROCESSOR_LIST will contain two instances of the host name where the tasks are running. This example removes any duplicate entries.

Note that LOADL_PROCESSOR_LIST is set by LoadLeveler, not by the user. This environment variable is limited to 128 hostnames. If the value is greater than the 128 limit, the environment variable is not set.

```
#!/bin/ksh
# @ output     =  my_POE_program.$(cluster).$(process).out
# @ error      =  my_POE_program.$(cluster).$(process).err
# @ class      =  POE
# @ job_type   =  parallel
# @ node = 8,12
# @ network.MPI = sn_single,shared,US
# @ queue

tmp_file="/tmp/node_list"
rm -f $tmp_file

# Copy each entry in the list to a new line in a file so
# that duplicate entries can be removed.
for node in $LOADL_PROCESSOR_LIST
        do
                echo $node >> $tmp_file
        done

# Sort the file removing duplicate entries and save list in variable
nodelist= sort -u /tmp/node_list

for node in $nodelist
        do
                rcp localfile $node:/home/userid
        done

rm -f $tmp_file


/usr/bin/poe /home/userid/my_POE_program
```

Building and submitting MPICH2 and serial interactive jobs

This topic describes how to submit MPICH2 and serial interactive LoadLeveler jobs.

To submit a LoadLeveler job, use the **llrun** command. Options are provided with the **llrun** command that allow:

- A command line to be specified
- Selected job characteristics to be specified
- A job command file to be specified

LoadLeveler uses either the job command file or the job characteristics option (the job command file and characteristics option are not permitted together) to allocate resources for the job and to start the specified command line option on the first node allocated for the job. The input or output for the command is directed to the console.

When a command line option is not specified, the **llrun** command will begin an interactive session on the first node in the list of nodes allocated to the interactive job. The **llrun** command uses LoadLeveler API calls to request resources for the interactive job and uses **ssh** to initiate your command or interactive session.

For more information about the **llrun** command, see the *LoadLeveler: Command and API Reference*.

Working with reservations

Under the BACKFILL scheduler only, LoadLeveler allows authorized users to make reservations, which specify a time period during which specific node resources are reserved for use by particular users or groups.

Use Table 44 to find information about working with reservations.

Table 44. Roadmap of tasks for reservation owners and users

Subtask	Associated instructions (see . . .)
Learn how reservations work in the LoadLeveler environment	• "Overview of reservations" on page 24 • "Types of reservations" • "Understanding the flexible job step" • "Understanding the reservation life cycle" on page 205
Creating new reservations	"Creating new reservations" on page 207
Managing jobs that run under a reservation	• "Submitting jobs to run under a reservation" on page 210 • "Removing bound jobs from the reservation" on page 212
Managing existing reservations	• "Querying existing reservations" on page 213 • "Modifying existing reservations" on page 213 • "Canceling existing reservations" on page 215
Managing floating resources	"Reservations with floating resources" on page 215
Using the LoadLeveler interfaces for reservations	Commands and Reservation API (see *LoadLeveler: Command and API Reference*)

Types of reservations

There are different types of reservations:

One-time reservation
A reservation with a specified start time and duration.

Recurring reservation
A reservation with a specified start time and duration that reoccurs for a specified period of time.

Flexible reservation
A reservation with a specified duration that will start as soon as the resources it requests become available. A flexible reservation cannot be made to recur.

Understanding the flexible job step

A flexible reservation will use a flexible job step to acquire resources for the reservation. If the flexible reservation is created with the **-f** option, the first job step in the specified job is used as the flexible job step; otherwise, LoadLeveler will create the flexible job step. LoadLeveler will submit the flexible job step when the reservation is being created. The job will not be dispatched by LoadLeveler. Instead, as soon as it has been scheduled, the resources acquired by it are transferred to the flexible reservation and the reservation becomes active. The flexible job step will then go into NOT_RUN state.

The following commands support flexible jobs:

- **llfavoruser**
- **llfavorjob**
- **llprio**

The following commands do not support flexible jobs:

- **llckpt**
- **llmodify**
- **llmovejob**
- **llpreempt**

Only an administrator can cancel a flexible job step. The flexible job step should not be canceled when the flexible job step is in IDLE state. The reservation can never become active if the flexible job step is canceled. If the reservation is canceled, then LoadLeveler will cancel the flexible job step.

If a failure occurs while creating a flexible reservation after the flexible job is submitted, the flexible job step will remain in the job queue with a flexible reservation ID of FLEXRES. During **llmkres** processing, the flexible job is submitted first with an ID of FLEXRES. When the reservation is created at the end of the command processing, it is filled with the reservation ID.

If the flexible reservation is created with the **-f** option and the job command file contains multiple steps, the first step will be the flexible job and it is not bound to the reservation. Subsequent steps will be bound to the flexible reservation. The **coschedule** and **data_stage** keywords cannot be used in the flexible job step.

If you do not specify the **-f** option when creating or modifying a flexible reservation, then LoadLeveler will create a temporary job command file. The temporary job command file contains the flexible job step in the current directory of the user executing the command using parameters specified in the **llmkres** or **llchres** command. The temporary flexible job step will be of the default class of the user creating the reservation. If no default class was set up for the user, the flexible job step will belong to the No_Class built-in class.

When you query a flexible job step by issuing the **llq -l** command, the long format will display the ID of the flexible reservation to which this job is attached or FLEXRES if it is not attached to a flexible reservation in the Flexible Res. ID field. The flexible reservation ID is the ID of the reservation to which this job is attached. For example:

```
[loadl@c197blade2b03 jcf]$ llq -l c197blade2b03.33.0
===== Job Step c197blade2b03.ppd.pok.ibm.com.33.0 =====
        Job Step Id: c197blade2b03.ppd.pok.ibm.com.33.0
           Job Name: c197blade2b03.ppd.pok.ibm.com.33
          Step Name: 0
  Structure Version: 10
              Owner: loadl
         Queue Date: Thu 10 Dec 2009 06:04:40 PM EST
             Status: Idle
    Reservation ID:
  Requested Res. ID:
   Flexible Res. ID: c197blade2b02.64.r
```

Understanding the reservation life cycle

From the time at which LoadLeveler creates a reservation through the time the reservation ends or is canceled, a reservation goes through various states, which are indicated in command listings and other displays or output. Understanding these states is important because the current state of a reservation dictates what actions you can take; for example, if you want to modify the start time for a reservation, you may do so only while the reservation is in Waiting state. Table 45 lists the possible reservation states, their abbreviations, and usage notes.

Table 45. Reservation states, abbreviations, and usage notes

Reservation state	Abbreviation in displays / output	Usage notes
Waiting	W	Reservations are in the Waiting state: 1. When LoadLeveler first creates a reservation. 2. After one occurrence of a recurring reservation ends and before the next occurrence starts. 3. While the flexible job step associated with a flexible reservation is in IDLE state. While the reservation is in the Waiting state: • Only administrators and reservation owners may modify, cancel, and add users or groups to the reservation. • Administrators, reservation owners, and users or groups that are allowed to use the reservation may query it, and submit jobs to run during the reservation period.
Setup	S	LoadLeveler changes the state of a reservation from Waiting to Setup just before the start time of the reservation. The actual time at which LoadLeveler places the reservation in Setup state depends on the value set for the **RESERVATION_SETUP_TIME** keyword in the configuration file. While the reservation is in Setup state: • Only administrators and reservation owners may modify, cancel, and add users or groups to the reservation. • Administrators, reservation owners, and users or groups that are allowed to use the reservation may query it, and submit jobs to run during the reservation period. During this setup period, LoadLeveler: • Stops scheduling unbound job steps to reserved nodes. • Preempts any jobs that are still running on the nodes that are reserved through this reservation. To preempt the running jobs, LoadLeveler uses the preemption method specified through the **DEFAULT_PREEMPT_METHOD** keyword in the configuration file. **Note:** The default value for **DEFAULT_PREEMPT_METHOD** is SU (suspend), which is not supported in all environments, and the default value for **PREEMPTION_SUPPORT** is NONE. If you want preemption to take place at the start of the reservation, make sure the cluster is configured for preemption (see "Steps for configuring a scheduler to preempt jobs" on page 126 for more information).

Table 45. Reservation states, abbreviations, and usage notes (continued)

Reservation state	Abbreviation in displays / output	Usage notes
Active	A	At the reservation start time, LoadLeveler changes the reservation state from Setup to Active. It also dispatches only job steps that are bound to the reservation, until the reservation completes or is canceled. For flexible reservations, once resources have been assigned to the associated flexible job step, LoadLeveler changes the reservation state from Waiting to Active. LoadLeveler does not dispatch bound job steps that: • Require certain resources, such as floating consumable resources, that are not available during the reservation period. • Have expected end times that exceed the end time of the reservation. By default, LoadLeveler allows such jobs to run, but their completion is subject to resource availability. (An administrator may configure LoadLeveler to prevent such jobs from running.) Jobs requesting floating resources that are bound to a reservation with assigned floating resources will not run if the expected end time for the job exceeds the end time of the reservation. These bound job steps remain idle unless the required resources become available. While the reservation is in Active state: • Only administrators and reservation owners may modify, cancel, and add users or groups to the reservation. • Administrators, reservation owners, and users or groups that are allowed to use the reservation may query it, and submit jobs to run during the reservation period.
Active_Shared	AS	At the reservation start time, LoadLeveler changes the reservation state from Setup to Active. It also dispatches only job steps that are bound to the reservation, unless the reservation was created with the `SHARED` mode. In this case, if reserved resources are still available after LoadLeveler dispatches any bound job steps that are eligible to run, LoadLeveler changes the reservation state to Active_Shared, and begins dispatching job steps that are not bound to the reservation. Once the reservation state changes to Active_Shared, it remains in that state until the reservation completes or is canceled. During this time, LoadLeveler dispatches both bound and unbound job steps, pending resource availability; bound job steps are considered before unbound job steps. The conditions under which LoadLeveler will not dispatch bound job steps are the same as those listed in the notes for the Active state. The actions that administrators, reservation owners, and users may perform are the same as those listed in the notes for the Active state.

Table 45. Reservation states, abbreviations, and usage notes (continued)

Reservation state	Abbreviation in displays / output	Usage notes
Canceled	CA	When a reservation owner, administrator, or LoadLeveler issues a request to cancel the reservation, LoadLeveler changes the state of a reservation to Canceled and unbinds any job steps bound to this reservation. When the reservation is in this state, no one can modify or submit jobs to this reservation. For flexible reservations, all running and idle jobs will be canceled once the reservation ends.
Complete	C	When a reservation end time is reached, LoadLeveler changes the state of a reservation to Complete. When the reservation is in this state, no one can modify or submit jobs to this reservation. For flexible reservations, all running and idle jobs will be canceled once the reservation ends.

Creating new reservations

You must be an authorized user or member of an authorized group to successfully create a reservation. LoadLeveler administrators define authorized users by adding the **max_reservations** keyword to the user or group stanza in the administration file. The **max_reservations** keyword setting also defines how many reservations you are allowed to own. Ask your administrator whether you are authorized to create reservations.

To be authorized to create reservations, LoadLeveler administrators also must have the **max_reservations** keyword set in their user or group stanza.

The **reservation_type** keyword in the user or group stanza defines what type of reservations can be created by an authorized user. LoadLeveler administrators can set it to **ALL**, **NONE**, or **FLEXIBLE** values. The default value is to be able to create all types of reservations.

To create a reservation, use the **llmkres** command. Specify the start time of the reservation using the **-t** command option and the duration of the reservation using the **-d** command option. If you are creating a recurring reservation, you must use the **-t** option to specify the schedule for that reservation.

To create a flexible reservation, use the **-x** option. The **-x** and **-t** flags are mutually exclusive.

In addition to the start time and duration (or reservation schedule), you must also use one of the following methods to specify how you want to select nodes for the reservation.

Note: These methods are mutually exclusive.

- The **-n** option on the **llmkres** command instructs LoadLeveler to reserve a number of nodes. LoadLeveler may select any unreserved node to satisfy a reservation. This command option is perhaps the easiest to use, because you need to know only how many nodes you want, not specific node characteristics.

  The minimum number of nodes a reservation must have is 1.

- The **-h** option on the **llmkres** command instructs LoadLeveler to reserve specific nodes.
- The **-f** option on the **llmkres** command instructs LoadLeveler to submit the specified job command file, and reserve appropriate nodes for the first job step in the job command file. Through this action, all job steps for the job are bound to the reservation. If the reservation request fails, LoadLeveler changes the state for all job steps for this job to NotQueued, and will not schedule any of those job steps to run.
- The **-j** option on the **llmkres** command instructs LoadLeveler to reserve appropriate nodes for that job step. Through this action, the job step is bound to the reservation. If the reservation request fails, the job step remains in the same state as it was before. This option is not supported for a flexible reservation.
- The **-c** option on the **llmkres** command instructs LoadLeveler to reserve a number of Blue Gene compute nodes (C-nodes). The **-j** and **-f** option also reserve Blue Gene resources if the job type is bluegene. This option is not supported for a flexible reservation.

You also may define other reservation attributes, including:

- Whether additional users or groups are allowed to use the reservation. Use the **-U** or **-G** command options, respectively.
- Whether the reservation will be in one or both of these optional modes:
  - `SHARED` mode: When you use the **-s** command option, LoadLeveler allows reserved resources to be shared by job steps that are not associated with a reservation. This mode enables the efficient use of reserved resources; if the bound job steps do not use all of the reserved resources, LoadLeveler can schedule unbound job steps as well so the resources do not remain idle. Unless you specify this mode, however, only job steps bound to the reservation may use the reserved resources. This option is not supported for a flexible reservation.
  - `REMOVE_ON_IDLE` mode: When you use the **-i** command option, LoadLeveler automatically cancels the reservation when all bound job steps that can run finish running. Using this mode is efficient because it prevents LoadLeveler from wasting reserved resources when no jobs are available to use them. Selecting this mode is especially useful for workloads that will run unattended.
- The default binding method to use when jobs are bound to the reservation. Use the **-m** option to specify whether the soft or firm binding method should be used when the binding method is not specified by the **llbind** command.
  - Soft binding allows the bound job to use resources outside of the reservation.
  - Firm binding restricts the job to the reserved resources.
  - A flexible reservation does not support soft binding.
- Floating resources for a reservation. Specify the resources using the **-C** option or use the **-f** option to specify a job command file with the **step_resources** or **resources** keyword. These resources must be configured as floating resources and specified in the **SCHEDULE_BY_RESOURCES** keyword in the LoadLeveler configuration.
- A notification program to be invoked whenever the reservation state changes. Use the **-p** option to specify the name of the user-supplied program. Two parameters, **Reservation ID** and **Reservation state**, will be passed to the user-specified notification program. For example, if you issue:

  ```
  llmkres -x -d 120 -f flexible_req.cmd -p '/u/loadl/notifyme'
  ```

  `/u/loadl/notifyme` is an example of an executable supplied by you.

Note: The user-supplied program must be in a file system that is accessible on the central manager node. The owner of the reservation must have execute permission for the program. In order to run a user-supplied script program, the first line has to specify the shell or the interpreter. Any environment variables required by the program will need to be set by the user-supplied program.

For a script:

$1 Is the reservation ID.

$2 Is the state.

For C programs:

arg[*0*] Is the program name

arg[*1*] Is the reservation ID.

arg[*2*] Is the state.

Sample notification program scripts:

The LoadLeveler samples directory contains sample scripts for how the notification program, submit notification program, and workflow engine work:

- On AIX, use directory **/usr/lpp/LoadL/scheduler/full/samples/workflow**
- On Linux, use directory **/opt/ibmll/LoadL/scheduler/full/samples/workflow**

- For a recurring reservation, when the reservation will expire. Use the **-e** option to specify the expiration date of the recurring reservation.
- For a flexible reservation, a limit for the amount of time the scheduler will try to acquire the resources required by the flexible reservation. Use the **-e** option to specify the expiration date. If the limit expires, then the reservation will be removed. If the **-e** flag is not specified in the **llmkres** command, the value of **max_reservation_expiration** in the user or group stanza in the configuration will be used as the expiration time for the flexible reservation. The value of **max_reservation_expiration** defaults to 180 days from now.
- For a flexible reservation, whether down nodes can be used for the reservation. Use the **-D** option to indicate that nodes may be down. The **-D** option can be used only with the **-h** or **-F** options or with the **-f** option if the job command file contains a host list. For example, issue:

```
llmkres -x -d 120 -D -f reservation_req.cmd
```

or:

```
llmkres -x -d 120 -D -h c80f1n01 c80f1n05 c80f1n08
```

Additional rules apply to the use of these options; see the **llmkres** command in *LoadLeveler: Command and API Reference* for details.

Alternative: Use the **ll_make_reservation** and the **ll_init_reservation_param** subroutines in a program.

Tips:

- If your user ID is not authorized to create any type of reservation but you are member of a group with authority to create reservations, you must use the **-g** option to specify the name of the authorized group on the **llmkres** command.
- Only reservations in waiting and in use are counted toward the limit of allowed reservations set through the **max_reservations** keyword. LoadLeveler does not count reservations or recurring reservations that have already ended or are in the process of being canceled.

- For accounting purposes, although recurring reservations have multiple instances, a recurring reservation counts as one reservation no matter how many times it may recur during its reservation period.
- Although you may create more than one reservation or recurring reservation for a particular node or set of nodes, only one of those reservations may be active at a time. If LoadLeveler determines that the reservation you are requesting will overlap with another reservation, LoadLeveler fails the create request. No reservation periods for the same set of machines can overlap.
- For a flexible reservation, the **RESERVATION_MIN_ADVANCE_TIME**, **RESERVATION_SETUP_TIME**, and **RESERVATION_PRIORITY** keywords are ignored.
- If the reservation requests floating consumable resources, there is a possibility that jobs running on nodes that are reserved for this request will also be preempted.

  Note: If the preemption method is suspend, then floating resources will not be released by the preempted job.

If the create request is successful, LoadLeveler assigns and returns to the owner a unique reservation identifier, in the form *host.rid*.r, where:

host The name of the machine which assigned the reservation identifier.

rid A number assigned to the reservation by LoadLeveler.

r The letter **r** is used to distinguish a reservation identifier from a job step identifier.

The following are examples of reservation identifiers:

```
c94n16.80.r
c94n06.1.r
```

For details about the LoadLeveler interfaces for creating reservations, see the **llmkres** command and the **ll_make_reservation** and **ll_init_reservation_param** subroutines in *LoadLeveler: Command and API Reference*.

Submitting jobs to run under a reservation

LoadLeveler administrators, reservation owners, and authorized users may submit jobs to run under a reservation. You may bind both batch and interactive POE job steps to a reservation, both before a reservation starts or while it is active.

Before you begin:

- If you are a reservation owner and used the **-f** or **-j** options on the **llmkres** command when you created the reservation, you do not have to perform the steps listed in Table 46 on page 211. Those command options automatically bind the job steps to the reservation. To find out whether a particular job step is bound to a reservation, use the command **llq -l** and check the listing for a reservation ID.
- To find out which reservation IDs you may use, check with your LoadLeveler administrator, or enter the command **llqres -l** and check the names in the Users or Groups fields (under the Modification time field) in the output listing. If your user name or a group name to which you belong appears in these output fields, you are authorized to use the reservation.
- LoadLeveler cannot guarantee that certain resources will be available during a reservation period. If you submit job steps that require these resources,

LoadLeveler will bind the job steps to the reservation, but will not dispatch them unless the resources become available during the reservation. These resources include:
  - Specific nodes that were not reserved under this reservation.
  - Floating consumable resources for a cluster.
  - Resources that are not released through preemption, such as virtual memory and adapters.
- Whether bound job steps are successfully dispatched depends not only on resource availability, but also on administration file keywords that set maximum numbers, including:
  - **max_jobs_scheduled**
  - **maxidle**
  - **maxjobs**
  - **maxqueued**

  If LoadLeveler determines that scheduling a bound job will exceed one or more of these configured limits, your job will remain idle unless conditions permit scheduling at a later time during the reservation period.

Table 46. Instructions for submitting a job to run under a reservation

To bind this type of job:	**Use these instructions:**
Already submitted jobs	Use the **llbind** command **Alternative:** Use the **ll_bind_reservation** subroutine in a program. **Result:** LoadLeveler either sets the reservation ID for each job step that can be bound to the reservation, or sends a failure notification for the bind request.
A new job that has not been submitted	1. Specify the reservation ID through the LL_RES_ID environment variable or the **ll_res_id** command file keyword. The **ll_res_id** keyword takes precedence over the LL_RES_ID environment variable. **Tip:** You can use the **ll_res_id** keyword to modify the reservation to submit to in a job command file filter. 2. Use the **llsubmit** command to submit the job. **Result:** If the job can be bound to the requested reservation, LoadLeveler sets the reservation ID for each job step that can be bound to the reservation. Otherwise, if the job step cannot be bound to the reservation, LoadLeveler changes the job state to NotQueued. To change the job step's state to Idle, issue the **llbind -r** command.

Use the **llqres** command or **llq** command with the **-l** option to check the success or failure of the binding request for each job step.

Selecting firm or soft binding: There are two methods by which a job step can be bound to a reservation: **firm** and **soft**. When a job step is firm bound to a reservation, the job step can only use the reserved resources. A job step that is soft bound to a reservation can be started before the reservation becomes active and can use nodes that are not part of the reservation. Using soft binding is a way of guaranteeing that resources will be available for the job step at a given time, but allowing the job step to start earlier if there are available resources. A flexible reservation does not support soft binding.

Which method to use is specified by the **-m** option of the **llbind** command. If neither is specified by **llbind**, the default method specified for the reservation is used. Use **llqres -l** and review the **Binding Method** field to determine which method is the default for a reservation.

Binding a job step to a recurring reservation: When a job step is bound to a reservation, the job step can be considered for scheduling as soon as any occurrence of the reservation is active. If you do not want the job step to run right away, but instead you want it to run in a later occurrence of the reservation, you can specify which occurrence the job step will be bound to by adding the occurrence ID to the end of the reservation ID.

The format of the reservation identifier is [*host*.]*rid*[.**r**[.*oid*]].

where:

- *host* is the name of the machine that assigned the reservation identifier.
- *rid* is the number assigned to the reservation when it was created. An *rid* is required.
- **r** indicates that this is a reservation ID (**r** is optional if *oid* is not specified).
- *oid* is the occurrence ID of a recurring reservation (*oid* is optional).

When *oid* is specified, the job step will not be considered for scheduling until that occurrence of the reservation becomes active. The step will remain in Idle state during all earlier occurrences.

If a job step is bound to a recurring reservation, and the reservation occurrence's end time is reached before the job step can be scheduled to run, the job step will be automatically bound to the next occurrence of the reservation by LoadLeveler. When the next occurrence becomes active, the job step will again be considered for scheduling.

A job can be submitted with the **recurring** keyword set to **yes** in the job command file to specify that all steps of the job will be run in every occurrence of the reservation to which it is bound. When all steps of the job have completed, the entire job is requeued and all steps are bound to the next occurrence of the reservation.

A flexible job is not bound to the flexible reservation because this job is used only to activate the reservation and it will never go into Running state.

For details about the LoadLeveler interfaces for submitting jobs under reservations, see the **llbind** command, the **ll_bind** subroutine, and the **llsubmit** command in *LoadLeveler: Command and API Reference*.

Removing bound jobs from the reservation

LoadLeveler administrators, reservation owners, and authorized users may use the **llbind** command to unbind one or more existing jobs from a reservation.

Alternative: Use the **ll_bind_reservation** subroutine in a program.

Result: LoadLeveler either unbinds the jobs from the reservation, or sends a failure notification for the unbind request. Use the **llqres** or **llq** command to check the success or failure of the remove request.

For details about the LoadLeveler interfaces for removing bound jobs from the reservation, see the **llbind** command and the **ll_bind** subroutine in *LoadLeveler: Command and API Reference*.

Querying existing reservations

Any LoadLeveler administrator or user can issue the **llqres** and **llq** commands to query the status of an existing reservation or recurring reservation.

Use these commands to request specific information about reservations:
- Various options are available to filter reservations to be displayed.
- To show details of specific reservations, use the **llqres** command with the **-l** option.
- To show job steps that are bound to specific reservations, use the **llq** command with the **-R** option.

When querying flexible reservations:
- The reservation type is FLEXIBLE.
- For a flexible reservation in Waiting state, the start time, end time, and node allocation are not set until the reservation becomes active.
- For a reservation in Waiting state, the **llqres -l** command output will display a section called, "Flexible Reservation Information", which will contain the user-specified node selection method used and the expiration time.
- For a reservation in Active state, the expiration time will not be displayed.
- The setup state will not be displayed.
- The **llqres -l** command will show the corresponding flexible job ID for this reservation.
- The **llqres -l** command will show the corresponding flexible reservation ID (Flexible Res. ID) for this job.

For details about reservation attributes and **llq** or **llqres** command syntax and examples, see *LoadLeveler: Command and API Reference*.

Modifying existing reservations

Only administrators and reservation owners can use the **llchres** command to modify one or more attributes of a reservation or a recurring reservation. Certain attributes cannot be changed after a reservation has become active. Typical uses for the **llchres** command include the following:
- Using the command **llchres -U *+newuser1 newuser2*** to allow additional users to submit jobs to the reservation.
- If a reservation was made through the command **llmkres -h free** but LoadLeveler cannot include a particular node because it is down, you can use the command **llchres -h *+node*** to add the node to the reserved node list when that node becomes available again.
- If a reserved node is down after the reservation becomes active, a LoadLeveler administrator can use:
  - The command **llchres -h *-node*** to remove that node from the reservation.
  - The command **llchres -h +1** to add another node to the reservation.
- Extending the expiration of a recurring reservation which may be about to expire. You can use **llchres -e** to specify a new expiration date for the reservation without having to create a new reservation.
- Making a temporary change to the next occurrence of a recurring reservation without affecting any future occurrences of that reservation. For example, you

can use the **-o** option of the **llchres** command to temporarily add a user (**-U**) or additional nodes (**-n**). Once that occurrence ends, the next occurrence will not retain the change.

Alternative: Use the **ll_change_reservation** subroutine in a program.

For details about the LoadLeveler interfaces for modifying reservations, see the **llchres** command and the **ll_change_reservation** subroutine in *LoadLeveler: Command and API Reference*.

Modifying flexible reservations

Only administrators and reservation owners can use the **llchres** command to modify one or more attributes of a flexible reservation. The attributes that cannot be changed after a reservation has become active for a one-time reservation also apply to a flexible reservation. There are some additional considerations that apply to flexible reservations:

- If node methods (**-n**, **-h**, **-F**, or **-f**), duration (**-d**), and floating consumable resources (**-C**) are changed, the flexible job step will be canceled and a new one will be submitted. It is possible that the new flexible job step will not have the same priority as the previous one.
- Floating resources cannot be modified after the reservation becomes active.
- In Waiting state, a modification can only be made if the creation method matches the modify method for the following:
  - The set of nodes
  - Duration
  - Floating resources

For example, if you issued:

```
llmkres -x -n 10 -d 130
```

and you want to modify the flexible reservation, issue:

```
llchres -x -n 2
```

and the modification will succeed.

But, if you issue:

```
llchres -x -f j.cmd
```

the modification will fail. The node selection method of **-n** does not match the change node selection method of **-f**.

To change the notification program of reservation c890f2ec01.19.r , issue:

```
llchres -p /home/llbld/jobs/myprog -R c890f2ec01.19.r
```

To change the floating resources of reservation c890f2ec01.20.r, issue:

```
llchres -C "MyArea(10)" -R c890f2ec01.20.r
```

Note: This is a full replacement of the original floating resources.

Canceling existing reservations

Administrators and reservation owners may use the **llrmres** command to cancel one or more reservations or to cancel some occurrences of a recurring reservation while leaving the remaining occurrences of that reservation unchanged in the system.

The options available when canceling a reservation are:

- Remove the entire reservation. All occurrences are removed and any bound job steps are automatically unbound from the reservation.
- Remove a specific occurrence of the reservation. All other occurrences remain in the system and all bound job steps remain bound to the reservation.
- Remove all occurrences during a specified interval. For example, a reservation may recur every day for one year, but during a one-week holiday period, the reservation is not needed. The reservation owner could cancel all of the occurrences during that one week period and all other occurrences would remain in the system and all bound job steps would remain bound to the reservation.

If some occurrences are canceled and the result is that no occurrences remain, then the entire reservation is removed and all jobs are unbound from the reservation.

For a flexible reservation, the flexible job and all jobs bound to the flexible reservation will be canceled when the reservation is removed.

Alternative: Use the **ll_remove_reservation** subroutine in a program.

Use the **llqres** command to check the success or failure of the remove request.

Use the **llqres -l** command to see a list of canceled occurrence IDs or to note individual occurrence start times which have been omitted due to cancellation.

For details about the LoadLeveler interfaces for canceling reservations, see the **llrmres** command and the **ll_remove_reservation** subroutine in *LoadLeveler: Command and API Reference*.

Reservations with floating resources

All types of reservations can reserve floating resources.

Jobs that are bound to a reservation are allocated floating consumable resources from the reservation rather than from the general pool of cluster-wide resources maintained by the scheduler.

If the reservation does not request floating resources, jobs bound to the reservation requiring floating resources will get the available resources from the cluster.

If the reservation requested floating resources and a job bound to the reservation requires more resources than those assigned to the reservation, the job will remain idle.

If a reservation requests floating resources, any bound jobs requiring floating resources will not start if the job's end time exceeds the reservation's end time. In this case, the **reservation_can_be_exceeded** keyword is ignored.

Requesting floating consumable resources

Floating consumable resources for a reservation can be requested with the **llmkres** or **llchres** command. For example:

To create a one-time reservation requesting one LicenseABC floating resource, issue:

```
llmkres -t 5/20/2010 8:00 -d 2 -n 2 -C "LicenseABC(1)"
```

To create a recurring reservation requesting two LicenseABC and four CompilerOptions floating resources, issue.

```
llmkres -t "10 7 * * * 05/20 07:10" -d 2 -n 1 -e 06/20 07:10 -C "LicenseABC(2) \
 CompilerOptionsB(4)"
```

To create a flexible reservation requesting one LicenseABC and three CompilerOptions floating resources, issue.

```
llmkres -x -d 2 -n 2 -C "LicenseABC(1) CompilerOptionsB(3)"
```

The flexible reservation will create a flexible job that will request floating resources for the flexible reservation using the **step_resources** keyword.

The **step_resources** keyword is the total floating resources for the job step used in the job command file. The syntax for the **step_resources** keyword is **step_resources=***name*(*count*) *name*(*count*). For example:

```
step_resources = Conslicense(4) ConsBW(10)
```

See "Job command file keyword descriptions" on page 335 for more information about the **step_resources** keyword.

To create a flexible reservation where the resource request is contained in the user-supplied job command file named **job.cmd**, issue.

```
llmkres -x -d 2  -f /home/llbld/job.cmd
```

The **-f** and **-C** option are mutually exclusive.

Modifying floating consumable resources

The **-C** option is used to modify floating consumable resources in a reservation. This will be a full replacement of current floating resources if any were previously requested. Modification of floating resources can only be done on reservations that are not in the Active state and that were created with the **-C** option. For example:

To modify the one-time reservation to request one LicenseABC floating resource, issue:

```
llchres -C "LicenseABC(1)" -R c197e325n01.ppd.pok.ibm.com.85.r
```

To modify a recurring reservation to request two LicenseABC and four CompilerOptions floating resources, issue:

```
llchres -C "LicenseABC(2) CompilerOptionsB(4)" -R c197e325n01.ppd.pok.ibm.com.86.r
```

To modify a flexible reservation requesting one LicenseABC and three CompilerOptions floating resources, issue:

```
llchres -C "LicenseABC(1) CompilerOptionsB(3)" -R c197e325n01.ppd.pok.ibm.com.87.r
```

The flexible reservation will cancel the existing flexible job step and submit a new flexible job with the new resource requirements. Only the non-JCF reservation node selection option can use the **-C** option.

To modify a flexible reservation where job.cmd will contain the **resources** keyword requesting one LicenseABC, issue.

```
llchres  -f /home/llbld/job.cmd -R c197e325n01.ppd.pok.ibm.com.88.r
```

Only a new job command file can change floating resources for reservations that were created by the job command file.

Removing floating consumable resources

Floating resources requested for a reservation are returned for general use when a reservation ends or expires. Any bound running jobs will be canceled to free up resources to give back to the global count. Any jobs that will run beyond the end of the reservation with floating resources will remain idle so the job will not need to be canceled.

For flexible reservations, all jobs will be canceled when the reservation ends.

Submitting jobs requesting scheduling affinity

You can request that a job use scheduling affinity by setting the **RSET** and **TASK_AFFINITY** job command file keywords.

Specify **RSET** with a value of:

- **RSET_MCM_AFFINITY** to have LoadLeveler schedule the job to machines where **RSET_SUPPORT** is enabled with a value of **RSET_MCM_AFFINITY**.
- *user_defined_rset* to have LoadLeveler schedule the job to machines where **RSET_SUPPORT** is enabled with a value of **RSET_USER_DEFINED**; *user_defined_rset* is the name of a valid user-defined RSet.

Specifying the **RSET** job command file keyword defaults to requesting memory affinity as a requirement and adapter affinity as a preference. Scheduling affinity options can be customized by using the job command file keyword **MCM_AFFINITY_OPTIONS**. For more information on these keywords, see "Job command file keyword descriptions" on page 335.

Note: If a job specifies memory or adapter affinity scheduling as a requirement, LoadLeveler will only consider machines where **RSET_SUPPORT** is set to **RSET_MCM_AFFINITY**. If there are not enough machines satisfying the memory affinity requirements, the job will stay in the idle state.

Specify **TASK_AFFINITY** with a value of:

- **CORE**(*n*) to have LoadLeveler schedule the job to machines where **RSET_SUPPORT** is enabled with a value of **RSET_MCM_AFFINITY**. On SMT and ST nodes, LoadLeveler will assign *n* physical CPUs to each job task.
- **CPU**(*n*) to have LoadLeveler schedule the job to machines where **RSET_SUPPORT** is enabled with a value of **RSET_MCM_AFFINITY**. On SMT nodes, LoadLeveler will assign *n* logical CPUs to each per job task. On ST nodes, LoadLeveler will assign *n* physical CPUs to each job task.

Specify a requirement of **SMT** with a value of:

- **Enabled** to have LoadLeveler schedule the job to machines where SMT is currently enabled.

  ```
  Example: #@ requirements = (SMT == "Enabled")
  ```

- **Disabled** to have LoadLeveler schedule the job to machines where SMT is currently disabled or is not supported.

  ```
  Example: #@ requirements = (SMT == "Disabled")
  ```

OpenMP multithreaded jobs can be submitted requesting thread-level binding, where each individual thread of an OpenMP application is bound to a separate physical core processor or logical CPU. Use the **parallel_threads** job command file keyword to request OpenMP thread-level binding, optionally, along with the **task_affinity** job command file keyword.

The CPUs to individual OpenMP threads of the tasks are selected based on the number of parallel threads (the **parallel_threads** job command file keyword) in each task and set of CPUs or cores assigned (the **task_affinity** job command file keyword) to the tasks. The CPUs are assigned to the threads only if at least one CPU is available for each thread from the set of CPUs or cores assigned to the task. If the number of CPUs in the set of CPUs or cores assigned to the tasks are not sufficient to bind all of the threads, the job will not run.

This example binds 4 OpenMP parallel threads to 4 separate cores:

```
#@ task_affinity = Core(4)
#@ parallel_threads = 4
```

Note: If you specify **cpus_per_core** along with your affinity request as:

```
#@ task_affinity = core(n)
#@ cpus_per_core = 1
```

Then LoadLeveler allocates the requested number of CPUs to each task on SMT nodes only. The nodes running in ST mode are not assigned for the jobs requesting **cpus_per_core**.

Submitting and monitoring jobs in a LoadLeveler multicluster

There are subtasks and associated instructions for submitting and monitoring jobs in a LoadLeveler multicluster.

Table 47 shows the subtasks and associated instructions for submitting and monitoring jobs in a LoadLeveler multicluster:

Table 47. Submitting and monitoring jobs in a LoadLeveler multicluster

Subtask	Associated instructions (see . . .)
Prepare and submit a job in the LoadLeveler multicluster	"Steps for submitting jobs in a LoadLeveler multicluster environment" on page 219
Display information about a job in the LoadLeveler multicluster environment	• Use the **llq -X** *cluster_name* command to display information about jobs on remote clusters. • Use **llq -x -d** to display the user's job command file keyword statements. • Use **llq -X** *cluster_name* **-l** to obtain multicluster specific information.

Table 47. Submitting and monitoring jobs in a LoadLeveler multicluster (continued)

Subtask	Associated instructions (see . . .)
Transfer an idle job from one cluster to another cluster	Use the **llmovejob** command, which is described in *LoadLeveler: Command and API Reference*.

Steps for submitting jobs in a LoadLeveler multicluster environment

There are steps for submitting jobs in a LoadLeveler multicluster environment.

In a multicluster environment, you can specify one of the following:
- That a job is to run on a particular cluster.
- That LoadLeveler is to decide which cluster is best from the list of clusters, based on an administrator-defined metric. If **any** is specified, the job is submitted to the best cluster, based on an administrator-defined metric.

The following procedure explains how to prepare your job to be submitted in the multicluster environment.

Before you begin: You need to know that:
- Only batch jobs are supported in the LoadLeveler multicluster environment. LoadLeveler will fail any interactive jobs that you attempt to submit in a multicluster environment.
- LoadLeveler assigns all steps of a multistep job to the same cluster.
- Job identifiers are assigned by the local cluster and are retained by the job regardless of what cluster the job executes in.
- Remote jobs are subjected to the same configuration checks as locally submitted jobs. Examples include account validation, class limits, include lists, and exclude lists.

Perform the following steps to submit jobs to run in one cluster in a LoadLeveler multicluster environment.

1. If files used by your job need to be copied between clusters, you must specify the job files to be copied from the local to the remote cluster in the job command file. Use the **cluster_input_file** and **cluster_output_file** keywords to specify these files.

   Rules:
   - Any local file specified for copy must be accessible from the local gateway Schedd machines. Input files must be readable. Directories and permissions must be in place to write output files.
   - Any remote file specified for copy must be accessible from the remote gateway Schedd machines. Directories and permissions must be in place to write input files. Output files must be readable when the job terminates.
   - To copy more than one file, these keywords can be specified multiple times.

   Tip: Each instance of these keywords allows you to specify a single local file and a single remote file. If your job requires copying multiple files (for example, all files in a directory), you may want to use a procedure to consolidate the multiple files into a single file rather than specify multiple cluster_file statements in the job command file. The following is an example of how you could consolidate input files:

   a. Use the **tar** command to produce a single tar file from multiple files.

b. On the **cluster_input_file** keyword, specify the file that resulted from the **tar** command processing.
c. Modify your job command file such that it uses the **tar** command to restore the multiple files from the tar file prior to invoking your application.

2. In the job command file, specify the clusters to which LoadLeveler may submit the job. The **cluster_list** keyword is a blank-delimited list of cluster names or the reserved word **any** where:
   - A single cluster name indicates that the job is to be submitted to that cluster.
   - A list of multiple cluster names indicates that the job is to be submitted to one of the clusters as determined by the installation exit **CLUSTER_METRIC**.
   - The reserved word **any** indicates that the job is to be submitted to any cluster defined by the installation exit **CLUSTER_METRIC**.

   Alternative: You can specify the clusters to which LoadLeveler can submit your job on the **llsubmit** command using the **-X** option.
3. Use the **llsubmit** command to submit the job.

   Tip: You may use the **-X** option on the **llsubmit** command to specify:
   -X {*cluster_list* | **any**}
   Is a blank-delimited list of cluster names or the reserved word **any** where:
   - A single cluster name indicates that the job is to be submitted to that cluster.
   - A list of multiple cluster names indicates that the job is to be submitted to one of the clusters as determined by the installation exit **CLUSTER_METRIC**.
   - The reserved word **any** indicates that the job is to be submitted to any cluster defined by the installation exit **CLUSTER_METRIC**.

   Note: If a remote job is submitted with a list of clusters or the reserved word **any** and the installation exit **CLUSTER_METRIC** is not specified, the remote job is not submitted.

The **llsubmit** command displays the assigned local outbound Schedd, the assigned remote inbound Schedd, the scheduling cluster and the job identifier when the remote job has been successfully submitted. Use the **-q** flag to stop these additional messages from being displayed.

When you are done, you can use commands to display information about the submitted job; for example:

- Use **llq -l -X** *cluster_name* **-j** *job_id* where *cluster_name* and *job_id* were displayed by the **llsubmit** command to display information about the remote job.
- Use **llq -l -X** *cluster_list* to display the long listing about jobs, including scheduling cluster, submitting cluster, user-requested cluster, cluster input and output files.
- Use **llq -X all** to display information about all jobs in all configured clusters.

Working with energy aware jobs

The energy policy tag (**energy_policy_tag**) helps LoadLeveler identify the energy data associated with a job. With the energy data, LoadLeveler can decide which frequency should be used to run the job with minimal performance degradation.

The energy policy tag identifies the energy associated with a job. The energy data includes:

- Power consumption and the elapsed time when run in the nominal frequency
- The estimated power consumption
- The elapsed time in other frequencies
- The percentage of performance degradation

When you set the energy policy tag in the job command file, the energy data is generated and stored in the database when the job runs for the first time. When the job is submitted again with the same energy policy tag, the same policy will be used. When you submit a job using the energy functions the first time, be sure to keep the energy tag name unique among the tags you have generated.

To set the energy keywords in the job command file to use the energy function, follow these steps:

1. Provide a unique identifier for the **energy_policy_tag** when a job is submitted the first time. For example:

   ```
   # user.cmd
   #@ energy_policy_tag = my_long_running_job
   ```

   LoadLeveler generates the energy data associated with this energy tag for the job when the job runs. To query the energy data using the energy tag, issue:

   ```
   llrqetag –e my_long_running_job
   ```

2. You can set an acceptable level of performance degradation in the job command file and resubmit the job to run at a lower energy level. Add the following to your job command file and submit it again:

   ```
   # user.cmd
   #@ energy_policy_tag = my_long_running_job
   #@ max_perf_decrease_allowed = 20
   ```

   After the job finishes, LoadLeveler will calculate the energy consumption for the job.

3. Issue the following command to get the energy consumption information for your jobs from the accounting data:

   ```
   llsummary -p
   ```

Submitting and monitoring Blue Gene jobs

This procedure explains how to prepare your job to be submitted to the Blue Gene system.

The submission of Blue Gene jobs is similar to the submission of other job types.

Before you begin: You need to know that checkpointing Blue Gene jobs is not currently supported.

Tip: Use the **llstatus** command to check if Blue Gene support is enabled and whether Blue Gene is currently present. The **llstatus** command will display:

```
The BACKFILL scheduler with Blue Gene support is in use

Blue Gene is present
```

when Blue Gene is support is enabled and Blue Gene is currently present.

Perform the following steps to submit Blue Gene jobs:

1. In the job command file, set the job type to Blue Gene by specifying:

```
#@job_type = bluegene
```

2. Specify the size or shape of the Blue Gene job or the Blue Gene block in which the job will run.
   - The size of the Blue Gene job can be specified by using the job command file keyword **bg_size** to specify the size of the job.
   - The shape of the Blue Gene job can be specified by using the job command file keyword **bg_shape** to specify the shape of the job. If you require the specific shape you specified, you may wish to specify the **bg_rotate** keyword to **false**.
   - The block in which the Blue Gene job is run can be specified using the **bg_block** job command file keyword.
   - The size of a Blue Gene job refers to the number of Blue Gene compute nodes instead of the number of tasks running on Startd machines. The following keywords cannot be used to control the size of a Blue Gene job:
     - **node**
     - **tasks_per_node**
     - **total_tasks**
3. Specify any other job command file keywords you require, including the **bg_connectivity** and **bg_requirements** Blue Gene job command file keywords.
4. Upon completing your job command file, submit the job using the **llsubmit** command.

For more information about any of the job command file keywords mentioned in this topic, see the detailed descriptions in "Job command file keyword descriptions" on page 335.

If you experience a problem submitting a Blue Gene job, see "Troubleshooting in a Blue Gene environment" on page 409 for common questions and answers pertaining to operations within a Blue Gene environment.

When you are done, you can use the **llq -b** command to display information about Blue Gene jobs in short form. For more information, see the **llq** command in *LoadLeveler: Command and API Reference*.

The following is a sample job command file for a Blue Gene job:

```
#!/bin/sh
# @ job_name           = bgsample
# @ job_type           = bluegene
# @ comment            = "BGQ Job By Size"
# @ error              = $(job_name).$(Host).$(Process).err
# @ output             = $(job_name).$(Host).$(Process).out
# @ executable         = /bgsys/drivers/ppcfloor/hlcs/bin/runjob
# @ arguments          = --exe /bgusr/loadl/myexe
# @ bg_size            = 1024
# @ wall_clock_limit   = 30:00
# @ bg_connectivity    = Torus
# @ queue
```

Chapter 8. Managing submitted jobs

This is a list of the tasks and sources of additional information for managing LoadLeveler jobs.

Table 48 lists the tasks and sources of additional information for managing LoadLeveler jobs.

Table 48. Roadmap of user tasks for managing submitted jobs

To learn about:	Read the following:
Displaying information about a submitted job or its environment	• "Querying the status of a job" • "Working with machines" • "Displaying currently available resources" on page 224 • **llclass**, **llq**, **llstatus**, and **llsummary** commands (see *LoadLeveler: Command and API Reference*)
Changing the priority of a submitted job	• "Setting and changing the priority of a job" on page 224 • **llmodify** command (see *LoadLeveler: Command and API Reference*)
Changing the state of a submitted job	• "Placing and releasing a hold on a job" on page 225 • "Canceling a job" on page 226 • **llcancel** and **llhold** commands (see *LoadLeveler: Command and API Reference*)
Checkpointing a submitted job	• "Checkpointing a job" on page 226 • **llckpt** command (see *LoadLeveler: Command and API Reference*)

Querying the status of a job

Once you submit a job, you can query the status of the job to determine, for example, if it is still in the queue or if it is running.

You also receive other job status related information such as the job ID and the submitting user ID. You can query the status of a LoadLeveler job by using the **llq** command. For an example of querying the status of a job, see Chapter 9, "Example: Using commands to build, submit, and manage jobs," on page 227.

Querying the status of a job using a submit-only machine: In addition to allowing you to submit and cancel jobs, a submit-only machine allows you to query the status of jobs. You can query a job by using the **llq** command. For information on the **llq** command, see *LoadLeveler: Command and API Reference*.

Working with machines

There are types of tasks related to machines.

You can perform the following types of tasks related to machines:

- **Display central manager**

The LoadLeveler administrator designates one of the machines in the LoadLeveler cluster as the central manager. When jobs are submitted to any machine, the central manager is notified and decides where to schedule the jobs. In addition, it keeps track of the status of machines in the cluster and jobs in the system by communicating with each machine. LoadLeveler uses this information to make the scheduling decisions and to respond to queries.

Usually, the system administrator is more concerned about the location of the central manager than the typical end user but you may also want to determine its location. One reason why you might want to locate the central manager is if you want to browse some configuration files that are stored on the same machine as the central manager.

- **Display public scheduling machines**

  Public scheduling machines are machines that participate in the scheduling of LoadLeveler jobs on behalf of users at submit-only machines and users at other workstations that are not running the Schedd daemon. You can find out the names of all these machines in the cluster.

  Submit-only machines allow machines that are not part of the LoadLeveler cluster to submit jobs to the cluster for processing.

Displaying currently available resources

The LoadLeveler user can get information about currently available resources by using the **llstatus** command with either the **-F**, or **-R** options.

The **-F** option displays a list of all of the floating resources associated with the LoadLeveler cluster. The **-R** option lists all of the consumable resources associated with all of the machines in the LoadLeveler cluster. The user can specify a hostlist with the **llstatus** command to display only the consumable resources associated with specific hosts.

Setting and changing the priority of a job

LoadLeveler uses the priority of a job to determine its position among a list of all jobs waiting to be dispatched.

LoadLeveler schedules jobs based on the *adjusted system priority*, which takes in account both system priority and user priority:

User priority

Every job has a user priority associated with it. A job with a higher priority runs before a job with a lower priority (when both jobs are owned by the same user). You can set this priority through the **user_priority** keyword in the job command file, and modify it through the **llprio** command. For more information on the **llprio** command, see *LoadLeveler: Command and API Reference*.

System priority

Every job has a system priority associated with it. Administrators can set this priority in the configuration file using the **SYSPRIO** keyword expression. The **SYSPRIO** expression can contain class, group, and user priorities, as shown in the following example:

```
SYSPRIO : (ClassSysprio * 100) + (UserSysprio * 10) + (GroupSysprio * 1) - (QDate)
```

The **SYSPRIO** expression is evaluated by LoadLeveler to determine the overall system priority of a job. To determine which jobs to run first, LoadLeveler does the following:

1. Assigns a system priority value when the negotiator adds the new job to the queue of jobs eligible for dispatch.
2. Orders jobs first by system priority.
3. Assigns jobs belonging to the same user and the same class an adjusted system priority, which takes all the system priorities and orders them by user priority. Jobs with a higher adjusted system priority are scheduled ahead of jobs with a lower adjusted system priority.

Only administrators may modify the system priority through the **llmodify** command with the **-s** option. For more information on the **llmodify** command, see *LoadLeveler: Command and API Reference*.

Example: How does a job's priority affect dispatching order?

To understand how a job's priority affects dispatching order, consider the sample jobs in Table 49, which lists the priorities assigned to jobs submitted by two users, Rich and Joe.

Two of the jobs belong to Joe, and three belong to Rich. User Joe has two jobs (Joe1 and Joe2) in Class A with SYSPRIOs of 9 and 8 respectively. Since Joe2 has the higher user priority (20), and because both of Joe's jobs are in the same class, Joe2's priority is swapped with that of Joe1 when the adjusted system priority is calculated. This results in Joe2 getting an adjusted system priority of 9, and Joe1 getting an adjusted system priority of 8. Similarly, the Class A jobs belonging to Rich (Rich1 and Rich3) also have their priorities swapped. The priority of the job Rich2 does not change, since this job is in a different class (Class B).

Table 49. How LoadLeveler handles job priorities

Job	User Priority	System Priority (SYSPRIO)	Class	Adjusted System Priority
Rich1	50	10	A	6
Joe1	10	9	A	8
Joe2	20	8	A	9
Rich2	100	7	B	7
Rich3	90	6	A	10

Placing and releasing a hold on a job

You may place a hold on a job and thereby cause the job to remain in the queue until you release it.

There are two types of holds: a user hold and a system hold. Both you and your LoadLeveler administrator can place and release a user hold on a job. Only a LoadLeveler administrator, however, can place and release a system hold on a job.

You can place a hold on a job or release the hold by using the **llhold** command. For examples of holding and releasing jobs, see Chapter 9, “Example: Using commands to build, submit, and manage jobs,” on page 227.

As a user or an administrator, you can also use the **startdate** keyword in “Job command file keyword descriptions” on page 335 to place a hold on a job. This keyword allows you to specify when you want to run a job.

Canceling a job

You can cancel one of your jobs that is either running or waiting to run by using the **llcancel** command. You can use **llcancel** to cancel LoadLeveler jobs, including jobs from a submit-only machine.

For more information about the **llcancel** command, see *LoadLeveler: Command and API Reference*.

Checkpointing a job

Checkpointing is a method of periodically saving the state of a job so that, if for some reason, the job does not complete, it can be restarted from the saved state. Checkpoints can be taken either under the control of the user application or external to the application.

To initiate a checkpoint from within a parallel application, use the API **mpc_init_ckpt**. This API allows the writer of the application to determine at what points in the application it would be appropriate save the state of the job. To enable parallel applications to initiate checkpointing, you must use the APIs provided with the Parallel Environment (PE) program. For information on parallel checkpointing, see *IBM Parallel Environment for AIX and Linux: Operation and Use, Volume 1*.

It is also possible to checkpoint a program running under LoadLeveler outside the control of the application. There are several ways to do this:

- Use the **llckpt** command to initiate checkpoint for a specific job step. For more information, see *LoadLeveler: Command and API Reference*.
- Checkpoint from a program which invokes the **ll_ckpt** API to initiate checkpoint of a specific job step. For more information, see *LoadLeveler: Command and API Reference*.
- Have LoadLeveler automatically checkpoint all running jobs that have been enabled for checkpoint. To enable this automatic checkpoint, specify **checkpoint = interval** in the job command file.
- As the result of an **llctl flush** command.

Note: For interactive parallel jobs, the environment variable **CHECKPOINT** must be set to **yes** in the environment prior to starting the parallel application or the job will not be enabled for checkpoint. For more information see, *IBM Parallel Environment for AIX and Linux: MPI Programming Guide*.

Chapter 9. Example: Using commands to build, submit, and manage jobs

This procedure presents a series of simple tasks that a user might perform using commands.

For additional information about individual commands noted in the procedure, see *LoadLeveler: Command and API Reference*.

1. Build your job command file by using a text editor to create a script file. Into the file enter the name of the executable, other keywords designating such things as output locations for messages, and the necessary LoadLeveler statements, as shown in the following example:

   ```
   # This job command file is called longjob.cmd.  The
   # executable is called longjob, the input file is longjob.in,
   # the output file is longjob.out, and the error file is
   # longjob.err.
   #
   # @ executable = longjob
   # @ input      = longjob.in
   # @ output     = longjob.out
   # @ error      = longjob.err

   # @ queue
   ```

2. You can optionally edit the job command file you created in step 1.
3. To submit the job command file that you created in step 1, use the **llsubmit** command:

   ```
   llsubmit longjob.cmd
   ```

   LoadLeveler responds by issuing a message similar to:

   ```
   submit: The job "wizard.22" has been submitted.
   ```

   Where `wizard` is the name of the machine to which the job was submitted and `22` is the job identifier (ID). You may want to record the identifier for future use (although you can obtain this information later if necessary).
4. To display the status of the job you just submitted, use the **llq** command. This command returns information about all jobs in the LoadLeveler queue:

   ```
   llq wizard.22
   ```

   Where `wizard` is the machine name to which you submitted the job, and `22` is the job ID. You can also query this job using the command **llq wizard.22.0**, where `0` is the step ID.
5. To change the priority of a job, use the **llprio** command. To increase the priority of the job you submitted by a value of 10, enter:

   ```
   llprio +10 wizard.22.0
   ```

   You can change the user priority of a job that is in the queue or one that is running. This only affects jobs belonging to the same user and the same class. If you change the priority of a job in the queue, the job's priority increases or decreases in relation to your other jobs in the queue. If you change the priority of a job that is running, it does not affect the job while it is running. It only affects the job if the job re-enters the queue to be dispatched again. For more information, see "Setting and changing the priority of a job" on page 224.

6. To place a temporary hold on a job in a queue, use the **llhold** command. This command only takes effect if jobs are in the Idle or NotQueued state. To place a hold on `wizard.22.0`, enter:

   ```
   llhold wizard.22.0
   ```

7. To release the hold you placed in step 6, use the **llhold** command:

   ```
   llhold -r wizard.22.0
   ```

8. To display the status of the machine to which you submitted a job, use the **llstatus** command:

   ```
   llstatus -l wizard
   ```

9. To cancel *wizard.22.0*, use the **llcancel** command:

   ```
   llcancel wizard.22.0
   ```

Part 4. LoadLeveler interfaces reference

The topics in the LoadLeveler interfaces reference provide the details you need to know to correctly use the IBM LoadLeveler interfaces for specifying keywords in the LoadLeveler control files.

Chapter 10. Configuration keyword reference

The configuration contains many parameters that you can set or modify to control how LoadLeveler operates.

For a file-based configuration, you can control LoadLeveler's operation either:

- Across the cluster, by modifying the global configuration file, **LoadL_config**, or
- Locally, by modifying the **LoadL_config.local** file on individual machines.

For a database-based configuration, you can control LoadLeveler's operation either:

- Across the cluster, by modifying values in tables for the cluster or for the default machine, or
- By modifying the records for individual machines in the database tables.

Table 50 shows the configuration subtasks:

Table 50. Configuration subtasks

Subtask	Associated information (see . . .)
To find out what administrator tasks you can accomplish by using the configuration	Chapter 4, "Configuring the LoadLeveler environment," on page 39
To learn how to correctly specify the contents of a configuration	• "Configuration keyword syntax" • "Configuration keyword descriptions" on page 233 • "User-defined keywords" on page 284 • "LoadLeveler variables" on page 286

Configuration keyword syntax

For files-based configuration, the information in both the **LoadL_config** and the **LoadL_config.local** files is in the form of a statement. These statements are made up of *keywords* and *values*.

There are three types of configuration keywords:

- Keywords, described in "Configuration keyword descriptions" on page 233.
- User-defined variables, described in "User-defined keywords" on page 284.
- LoadLeveler variables, described in "LoadLeveler variables" on page 286.

Configuration file statements take one of the following formats:

```
keyword=value
keyword:value
```

Statements in the form *keyword=value* are used primarily to customize an environment. Statements in the form *keyword:value* are used by LoadLeveler to specify expressions. For example, the **SYSPRIO** keyword specifies the expression used to calculate the system priority for job steps:

```
SYSPRIO: 0 - QDate
```

Keywords are *not* case sensitive. This means you can enter them in lower case, upper case, or mixed case.

Note: For the *keyword=value* form, if the keyword is of a boolean type and only **true** and **false** are valid input, a value string starting with **t** or **T** is taken as **true**; all other values are taken as **false**.

To continue configuration file statements, use the back-slash character (\).

In the configuration file, comments must be on a separate line from keyword statements.

You can use the following types of constants and operators in configuration keywords.

Numerical and alphabetical constants

These are the numerical and alphabetical constants.

Constants may be represented as:
- Boolean expressions
- Signed integers
- Floating point values
- Strings enclosed in double quotes (" ").

Mathematical operators

You can use the following C operators.

The operators are listed in order of precedence. All of these operators are evaluated from left to right:
- !
- * /
- - +
- < <= > >=
- == !=
- &&
- ||

64-bit support for configuration file keywords and expressions

Administrators can assign 64-bit integer values to selected configuration keywords.

floating_resources
Consumable resources associated with the **floating_resources** keyword may be assigned 64-bit integer values. Fractional and unit specifications are not allowed. The predefined ConsumableCpus, ConsumableMemory, ConsumableLargePageMemory, and ConsumableVirtualMemory may not be specified as floating resources.

Example:

```
floating_resources = spice2g6(9876543210123) db2_license(1234567890)
```

MACHPRIO expression
The LoadLeveler variables: Disk, ConsumableCpus, ConsumableMemory, ConsumableVirtualMemory, ConsumableLargePageMemory, PagesScanned, Memory, VirtualMemory, FreeRealMemory, and PagesFreed may be used in a MACHPRIO expression. They are 64-bit integers and 64-bit arithmetic is used to evaluate them.

Example:

```
MACHPRIO: (Memory + FreeRealMemory) - (LoadAvg*1000 + PagesScanned)
```

Configuration keyword descriptions

This topic provides an alphabetical list of the keywords you can use in a LoadLeveler configuration.

It also provides examples of statements that use these keywords.

ACCT

Turns the accounting function on or off.

Syntax:

```
ACCT = flag ...
```

The available flags are:

A_DETAIL
Enables extended accounting. Using this flag causes LoadLeveler to record detail resource consumption by machine and by events for each job step. This flag also enables the **-x** flag of the **llq** command, permitting users to view resource consumption for active jobs.

A_ENERGY
Turns energy data recording on.

A_RES
Turns reservation data recording on.

A_OFF
Turns accounting data recording off.

A_ON Turns accounting data recording on. If specified without the **A_DETAIL** flag, the following is recorded:
- The total amount of CPU time consumed by the entire job
- The maximum memory consumption of all tasks (or nodes).

A_VALIDATE
Turns account validation on.

Default value: A_OFF

Examples:

This example specifies that accounting should be turned on and that extended accounting data should be collected and that the -x flag of the **llq** command be enabled.

```
ACCT = A_ON A_DETAIL
```

This example specifies that accounting should be turned on and that extended accounting data and energy consumption data should be collected.

```
ACCT = A_ON A_DETAIL A_ENERGY
```

ACCT_VALIDATION

Identifies the executable called to perform account validation.

Syntax:

```
ACCT_VALIDATION = program
```

Where *program* is a validation program.

Default value: $(BIN)/llacctval (the accounting validation program shipped with LoadLeveler.

ACTION_ON_MAX_REJECT
Specifies the state in which jobs are placed when their rejection count has reached the value of the **MAX_JOB_REJECT** keyword. HOLD specifies that jobs are placed in User Hold status; SYSHOLD specifies that jobs are placed in System Hold status; CANCEL specifies that jobs are canceled. When a job is rejected, LoadLeveler sends a mail message stating why the job was rejected.

Syntax:

```
ACTION_ON_MAX_REJECT = HOLD | SYSHOLD | CANCEL
```

Default value: HOLD

ACTION_ON_SWITCH_TABLE_ERROR
Points to an administrator supplied program that will be run when **DRAIN_ON_SWITCH_TABLE_ERROR** is set to **true** and a switch table unload error occurs.

Syntax:

```
ACTION_ON_SWITCH_TABLE_ERROR = program
```

Default value: The default is to not run a program.

ADAPTER_HEARTBEAT_INTERVAL
Specifies the amount of time, in seconds, that defines the heartbeat interval between the region manager and startd. This keyword is used by the resource manager component only.

Syntax:

```
ADAPTER_HEARTBEAT_INTERVAL = interval
```

where:

interval is in seconds.

Default value: The default is 30 seconds. If a value of 0 is specified, the default will be used.

ADAPTER_HEARTBEAT_PORT
Specifies the port number on which the region manager listens for heartbeats from startd. This keyword is used by the resource manager component only.

Syntax:

```
ADAPTER_HEARTBEAT_PORT = port
```

where:

port is a positive whole number.

Default value: The default is 9684.

ADAPTER_HEARTBEAT_RETRIES
Specifies the number of heartbeat intervals that the region manager will wait before declaring the adapter on startd as down. This keyword is used by the resource manager component only.

Syntax:

```
ADAPTER_HEARTBEAT_RETRIES = retries
```

where:

retries is a positive whole number.

Default value: The default is 2. If a value of 0 is specified, the default will be used.

ADMIN_FILE
Points to the administration file containing user, class, group, machine_group, machine, cluster, and region stanzas.This keyword is not used for the database configuration option.

Syntax:

```
ADMIN_FILE = directory
```

Default value: `$(tilde)/admin_file`

AFS_GETNEWTOKEN
Specifies a filter that, for example, can be used to refresh an AFS token.

Syntax:

```
AFS_GETNEWTOKEN = full_path_to_executable
```

Where *full_path_to_executable* is an administrator-supplied program that receives the AFS authentication information on standard input and writes the new information to standard output. The filter is run when the job is scheduled to run and can be used to refresh a token which expired when the job was queued.

Default value: The default is to not run a program.

AGGREGATE_ADAPTERS
Allows an external scheduler to specify per-window adapter usages.

Syntax:

```
AGGREGATE_ADAPTERS = YES | NO
```

When this keyword is set to **YES**, the resources from multiple switch adapters on the same switch network are treated as one aggregate pool available to each job. When this keyword is set to **NO**, the switch adapters are treated individually and a job cannot use resources from multiple adapters on the same network.

Set this keyword to **NO** when you are using an external scheduler; otherwise, set to **YES** (or accept the default).

Default value: YES

ARCH
Indicates the standard architecture of the system. The architecture you specify here must be specified in the same format in the **requirements** and **preferences** statements in job command files. The administrator defines the character string for each architecture.

Syntax:

```
ARCH = string
```

Default value: Use the command **llstatus -l** to view the default.

Example: To define a machine as an RS/6000®, the keyword would look like:

```
ARCH = R6000
```

BG_ALLOW_LL_JOBS_ONLY
Specifies if only jobs submitted through LoadLeveler will be accepted by the Blue Gene job launcher program.

Syntax:

```
BG_ALLOW_LL_JOBS_ONLY = true | false
```

Default value: false

BG_CACHE_BLOCKS
Specifies whether allocated blocks are to be reused for Blue Gene jobs whenever possible.

Syntax:
```
BG_CACHE_BLOCKS = true | false
```

Default value: true

BG_ENABLE_PASSTHROUGH
Specifies whether LoadLeveler should consider "pass-through" midplanes when scheduling. This will allow for scheduling solutions using midplanes that are not topologically next to each other.

Syntax:
```
BG_ENABLE_PASSTHROUGH = true | false
```

Default value: false

BG_ENABLED
Specifies whether Blue Gene support is enabled.

Syntax:
```
BG_ENABLED = true | false
```

If the value of this keyword is **true**, the central manager will load the Blue Gene control system libraries and query the state of the Blue Gene system so that jobs of type **bluegene** can be scheduled.

Default value: false

BG_MIN_BLOCK_SIZE
Specifies the smallest number of compute nodes in a block.

Syntax:
```
BG_MIN_BLOCK_SIZE = 32 | 64 | 128 | 256 | 512
```

The value for this keyword must not be smaller than the minimum block size supported by the physical Blue Gene hardware. If the number of compute nodes requested by the job is less than the minimum block size, then LoadLeveler will increase the requested size to the minimum block size.

Note: The minimum block size is also limited by the Blue Gene System I/O ratio. If the system only has a I/O ratio of 1:128, the minimum block size will be set by LoadLeveler to 128, regardless of the value specified by **BG_MIN_BLOCK_SIZE**.

Default value: 32

BIN
Defines the directory where LoadLeveler binaries are kept.

Syntax:
```
BIN = $(RELEASEDIR)/bin
```

Default value: $(tilde)/bin

CENTRAL_MANAGER_HEARTBEAT_INTERVAL
Specifies the amount of time, in seconds, that defines how frequently the primary and alternate central manager communicate with each other.

Note: This keyword is deprecated in favor of the **FAILOVER_HEARTBEAT_INTERVAL** keyword.

Syntax:

```
CENTRAL_MANAGER_HEARTBEAT_INTERVAL = number
```

Default value: The default value is 300 seconds or 5 minutes.

CENTRAL_MANAGER_LIST

Specifies the list of machines where the primary and alternate central manager daemons run. This keyword overrides the **central_manager** statement in the machine stanza in the administration file. This keyword is used by the scheduler component only.

Syntax:

```
CENTRAL_MANAGER_LIST = primary_central_manager_machine \
[alternate_central_manager_machine_list]
```

Where *primary_central_manager_machine* is the host name of the machine that the primary central manager daemon will run on and *alternate_central_manager_machine_list* is a blank delimited list of host names for the alternate central manager daemons.

Default value: No default value is set.

CENTRAL_MANAGER_TIMEOUT

Specifies the number of heartbeat intervals that an alternate central manager will wait before declaring that the primary central manager is not operating.

Syntax:

```
CENTRAL_MANAGER_TIMEOUT = number
```

Default value: The default is 6.

Note: This keyword is deprecated in favor of the **FAILOVER_HEARTBEAT_RETRIES** keyword.

CKPT_CLEANUP_INTERVAL

Specifies the interval, in seconds, at which the **Schedd** daemon will run the program specified by the **CKPT_CLEANUP_PROGRAM** keyword.

Syntax:

```
CKPT_CLEANUP_INTERVAL = number
```

number must be a positive integer.

Default value: -1

CKPT_CLEANUP_PROGRAM

Identifies an administrator-provided program which is to be run at the interval specified by the **ckpt_cleanup_interval** keyword. The intent of this program is to delete old checkpoint files created by jobs running under LoadLeveler during the checkpoint process.

Syntax:

```
CKPT_CLEANUP_PROGRAM = program
```

Where *program* is the fully qualified name of the program to be run. The program must be accessible and executable by LoadLeveler.

A sample program to remove checkpoint files is provided in the `/usr/lpp/LoadL/full/samples/llckpt/rmckptfiles.c` file.

Default value: No default value is set.

CKPT_EXECUTE_DIR

Specifies the directory where the job step's executable will be saved for checkpointable jobs. You can specify this keyword in either the configuration keyword or the job command file; different file permissions are required depending on where this keyword is set. When used as a configuration keyword, it specifies a "base" directory. For each job step, a subdirectory with the name *job_step_id* is created and the job step's executable is copied to this subdirectory. For additional information, see "Planning considerations for checkpointing jobs" on page 136.

Syntax:

```
CKPT_EXECUTE_DIR = directory
```

This directory cannot be the same as the current location of the executable file, or LoadLeveler will not stage the executable. In this case, the user must have execute permission for the current executable file.

Default value: By default, the executable of a checkpointable job step is not staged.

Note: The staged executables are not deleted because they may be needed for restart operations. It is your responsibility to manage the files in the **ckpt_execute_dir** directory and remove any files that are no longer needed.

CLASS

Determines whether a machine will accept jobs of a certain job class. For parallel jobs, you must define a class instance for each task you want to run on a node using one of two formats:

- The format, **CLASS** = *class_name (count)*, defines the **CLASS** names using a statement that names the classes and sets the number of tasks for each class in parenthesis.

  With this format, the following rules apply:
  - Each class can have only one entry
  - If a class has more than one entry or there is a syntax error, the entire **CLASS** statement will be ignored
  - If the **CLASS** statement has a blank value or is not specified, it will be defaulted to **No_Class (1)**
  - The number of instances for a class specified inside the parenthesis **( )** must be an unsigned integer. If the number specified is 0, it is correct syntactically, but the class will not be defined in LoadLeveler
  - If the number of instances for all classes in the **CLASS** statement are 0, the default **No_Class(1)** will be used
- The format, **CLASS** = { "*class1*" "*class2*" "*class2*" "*class2*"}, defines the **CLASS** names using a statement that names each class and sets the number of tasks for each class based on the number of times that the class name is used inside the **{}** operands.

Note: With both formats, the class names list is blank delimited.

For a LoadLeveler job to run on a machine, the machine must have a vacancy for the class of that job. If the machine is configured for only one **No_Class** job and a LoadLeveler job is already running there, then no further LoadLeveler jobs are started on that machine until the current job completes.

You can have a maximum of 1024 characters in the class statement. You cannot use **allclasses** or **data_stage** as a class name, since these are reserved LoadLeveler keywords.

You can assign multiple classes to the same machine by specifying the classes in the LoadLeveler configuration file (called **LoadL_config**) or in the local configuration file (called **LoadL_config.local**). The classes, themselves, should be defined in the administration file. See "Setting up a single machine to have multiple job classes" on page 415 and "Defining classes" on page 94 for more information on classes.

Syntax:

```
CLASS = { "class_name" ... } | {"No_Class"} | class_name (count) ...
```

Default value: {"No_Class"}

Note: This keyword is deprecated in the configuration file and has been moved to the administration file.

CLIENT_TIMEOUT

Specifies the maximum time, in seconds, that a daemon waits for a response over TCP/IP from a process. If the waiting time exceeds the specified amount, the daemon tries again to communicate with the process. In general, you should use the default setting unless you are experiencing delays due to an excessively loaded network. If so, you should try increasing this value.

Syntax:

```
CLIENT_TIMEOUT = number
```

Default value: The default is 30 seconds.

CLUSTER_METRIC

Indicates the installation exit to be run by the Schedd to determine where a remote job is distributed. If a remote job is submitted with a list of clusters or the reserved word **any** and the installation exit is not specified, the remote job is not submitted.

Syntax:

```
CLUSTER_METRIC = full_pathname_to_executable
```

The installation exit is run with the following parameters passed as input. All parameters are character strings.

- The job ID of the job to be distributed
- The number of clusters in the list of clusters
- A blank-delimited list of clusters to be considered

If the user specifies the reserved word **any** as the **cluster_list** during job submission, the job is sent to the first outbound Schedd defined for the first configured remote cluster. The **CLUSTER_METRIC** is executed on this machine to determine where the job will be distributed. If this machine is not the **outbound_hosts** Schedd for the assigned cluster, the job will be forwarded to the correct **outbound_hosts** Schedd. If the user specifies a list of clusters as the **cluster_list** during job submission, the job is sent to the first outbound Schedd defined for the first specified remote cluster. The **CLUSTER_METRIC** is executed on this machine to determine where the job will be distributed. If this machine is not the **outbound_hosts** Schedd for the assigned cluster, the job will be forwarded to the correct **outbound_hosts** Schedd.

Note: The list of clusters may contain a single entry of the reserved word **any**, which indicates that the **CLUSTER_METRIC** installation exit must determine its own list of clusters to select from. This can be all of the clusters available using the data access API or a predetermined list set by the administrator. If **any** is specified in place of a cluster list, the metric will receive a count of 1

followed by the keyword **any**.
The installation exit must write the remote cluster name to which the job is submitted as standard output and exit with a value of 0. An exit value of -1 indicates an error in determining the cluster for distribution and the job is not submitted. Returned cluster names that are not valid also cause the job to be not submitted. STDERR from the exit is written to the Schedd log.

LoadLeveler provides a set of sample exits for use in distributing jobs by the following metrics:

- The number of jobs in the idle queue
- The number of jobs in the specified class
- The number of free nodes in the cluster

The installation exit samples are available in the **${RELEASEDIR}/samples/llcluster** directory.

CLUSTER_REMOTE_JOB_FILTER
Indicates the installation exit to be run by the inbound Schedd for each remote job request to filter the user's job command file statements during submission or move job. If the keyword is not specified, no job filtering is done.

Syntax:

```
CLUSTER_REMOTE_JOB_FILTER = full_pathname_to_executable
```

The installation exit is run with the submitting user's ID. All parameters are character strings.

This installation exit is executed on the **inbound_hosts** of the local cluster when receiving a job submission or move job request.

The executable specified is called with the submitting user's unfiltered job command file statements as the standard input. The standard output is submitted to LoadLeveler. If the exit returns with a nonzero exit code, the remote job submission or job move will fail. A submit filter can only make changes to LoadLeveler job command file statements.

The data access API can be used by the remote job filter to query the Schedd for the job object received from the sending cluster.

If the local submission filter on the submitting cluster has added or deleted steps from the original user's job command file, the remote job filter must add or delete the same number of steps. The job command file statements returned by the remote job filter must contain the same number of steps as the job object received from the sending cluster.

Changes to the following job command file keyword statements are ignored:

- **executable**
- **environment**
- **image_size**
- **cluster_input_file**
- **cluster_output_file**
- **cluster_list**

The following job command file keyword will have different behavior:

- **initialdir** – If not set by the remote job filter or the submitting user's unfiltered job command file, the default value will remain the current working directory at the time the job was submitted. Access to the **initialdir**

will be verified on the cluster selected to run the job. If access to **initialdir** fails, the submission or move job will fail.

To maintain compatibility between the **SUBMIT_FILTER** and **CLUSTER_REMOTE_JOB_FILTER** programs, the following environment variables are set when either exit is invoked:

- **LOADL_ACTIVE** – the LoadLeveler version.
- **LOADL_STEP_COMMAND** – the location of the job command file passed as input to the program. This job command file only contains LoadLeveler keywords.
- **LOADL_STEP_ID** – The job identifier, generated by the submitting LoadLeveler cluster.

  Note: The environment variable name is **LOADL_STEP_ID** although the value it contains is a "job" identifier. This name is used to be compatible with the local job filter interface.
- **LOADL_STEP_OWNER** – The owner (UNIX user name) of the job.

CLUSTER_USER_MAPPER

Indicates the installation exit to be run by the inbound Schedd for each remote job request to determine the user mapping of the cluster. This keyword implies that user mapping is performed. If the keyword is not specified, no user mapping is done.

Syntax:

```
CLUSTER_USER_MAPPER = full_pathname_to_executable
```

The installation exit is run with the following parameters passed as input. All parameters are character strings.

- The user name to be mapped
- The cluster name where the user originated from

This installation exit is executed on the **inbound_hosts** of the local cluster when receiving a job submission, move job request or remote command.

The installation exit must write the new user name as standard output and exit with a value of 0. An exit value of -1 indicates an error and the job is not submitted. STDERR from the exit is written to the Schedd log. An exit value of 1 indicates that the user name returned for this job was **not** mapped.

CM_CHECK_USERID

Specifies whether the central manager will check the existence of user IDs that sent requests through a command or API on the central manager machine.

Syntax:

```
CM_CHECK_USERID = true | false
```

Default value: true

CM_COLLECTOR_PORT

Specifies the port number used when connecting to a daemon.

Syntax:

```
CM_COLLECTOR_PORT = port number
```

Default value: The default is 9612.

COMM

Specifies a local directory where LoadLeveler keeps special files used for UNIX domain sockets for communicating among LoadLeveler daemons running on

the same machine. This keyword allows the administrator to choose a different file system other than /tmp for these files. If you change the COMM option you must stop and then restart LoadLeveler using the **llctl** command.

Syntax:

```
COMM = local directory
```

Default value: The default location for the files is **/tmp**.

CONTINUE
Determines whether suspended jobs should continue execution.

Syntax:

```
CONTINUE: expression that evaluates to T or F (true or false)
```

When **T**, suspended LoadLeveler jobs resume execution on the machine.

Default value: No default value is set.

For information about time-related variables that you may use for this keyword, see "Variables to use for setting times" on page 291.

CUSTOM_METRIC
Specifies a machine's relative priority to run jobs.

Syntax:

```
CUSTOM_METRIC = number
```

This is an arbitrary number which you can use in the MACHPRIO expression. Negative values are not allowed.

Default value: If you specify neither **CUSTOM_METRIC** nor **CUSTOM_METRIC_COMMAND**, **CUSTOM_METRIC = 1** is assumed. For more information, see "Setting negotiator characteristics and policies" on page 47.

For more information related to using this keyword, see "Defining a LoadLeveler cluster" on page 45.

CUSTOM_METRIC_COMMAND
Specifies an executable and any required arguments. The exit code of this command is assigned to **CUSTOM_METRIC**. If this command does not exit normally, **CUSTOM_METRIC** is assigned a value of 1. This command is forked every (**POLLING_FREQUENCY** * **POLLS_PER_UPDATE**) period.

Syntax:

```
CUSTOM_METRIC_COMMAND = command
```

Default value: No default is set; LoadLeveler does not run any command to determine CUSTOM_METRIC.

DCE_AUTHENTICATION_PAIR
Specifies a pair of installation supplied programs that are used to authenticate DCE security credentials.

Restriction: DCE security is not supported by LoadLeveler for Linux.

Syntax:

```
DCE_AUTHENTICATION_PAIR = program1, program2
```

Where *program1* and *program2* are LoadLeveler- or installation-supplied programs that are used to authenticate DCE security credentials. *program1* obtains a handle (an opaque credentials object), at the time the job is

submitted, which is used to authenticate to DCE. *program2* uses the handle obtained by *program1* to authenticate to DCE before starting the job on the executing machines.

Default value: See "Handling DCE security credentials" on page 77 for information about defaults.

DEFAULT_PREEMPT_METHOD

Specifies the default preemption method for LoadLeveler to use when a preempt method is not specified in a PREEMPT_CLASS statement or in the **llpreempt** command. LoadLeveler also uses this default preemption method to preempt job steps that are running on reserved machines when a reservation period begins.

Restrictions:

- This keyword is valid only for the BACKFILL scheduler.
- The suspend method of preemption (the default) might not be supported on your level of Linux. If you want to preempt jobs that are running where process tracking is not supported, you must use this keyword to specify a method other than suspend.

Syntax:

```
DEFAULT_PREEMPT_METHOD =  rm | sh | su | vc | uh
```

Valid values are:

rm LoadLeveler preempts the jobs and removes them from the job queue. To rerun the job, the user must resubmit the job to LoadLeveler.

sh LoadLeveler ends the jobs and puts them into System Hold state. They remain in that state on the job queue until an administrator releases them. After being released, the jobs go into Idle state and will be rescheduled to run as soon as resources for the job are available.

su LoadLeveler suspends the jobs and puts them in Preempted state. They remain in that state on the job queue until the preempting job has terminated, and resources are available to resume the preempted job on the same set of nodes. To use this value, process tracking must be enabled.

vc LoadLeveler ends the jobs and puts them in Vacate state. They remain in that state on the job queue and will be rescheduled to run as soon as resources for the job are available.

uh LoadLeveler ends the jobs and puts them into User Hold state. They remain in that state on the job queue until an administrator releases them. After being released, the jobs go into Idle state and will be rescheduled to run as soon as resources for the job are available.

Default value: **su** (suspend method)

For more information related to using this keyword, see "Steps for configuring a scheduler to preempt jobs" on page 126.

DRAIN_ON_SWITCH_TABLE_ERROR

Specifies whether the **startd** should be drained when the switch table fails to unload. This will flag the administrator that intervention may be required to unload the switch table. When **DRAIN_ON_SWITCH_TABLE_ERROR** is set to true, the **startd** will be drained when the switch table fails to unload.

Syntax:

```
DRAIN_ON_SWITCH_TABLE_ERROR = true | false
```

Default value: false

DSTG_MAX_STARTERS

Specifies a machine-specific limit on the number of data staging initiators. Since each task of a data staging job step consumes one initiator from the **data_stage** class on the specified machine, **DSTG_MAX_STARTERS** provides the maximum number of data staging tasks that can run at the same time on the machine.

Syntax:

```
DSTG_MAX_STARTERS = number
```

Notes:

1. If you have not set the **DSTG_MAX_STARTERS** value in either the global or local configuration files, there will not be any data staging initiators on the specified machine. In this configuration, the executing machine will not be allowed to perform data staging tasks.
2. The value specified for **DSTG_MAX_STARTERS** will be the number of initiators available for the built-in **data_stage** class on that machine.
3. The value specified for **MAX_STARTERS** will not limit the value specified for **DSTG_MAX_STARTERS**.

Default value: 0

Note: This keyword is deprecated in the configuration file and has been moved to the administration file.

DSTG_MIN_SCHEDULING_INTERVAL

Specifies a minimum interval between scheduling inbound data staging job steps when they cannot be scheduled immediately. With a workload that involves a lot of data staging jobs, this keyword can be adjusted down from the default value of 900 seconds, if data staging jobs remain idle when there are data staging resources available. Setting this keyword to a smaller interval may impact scheduler performance when there is contention for data staging resources and a large number of idle jobs in the queue.

Syntax:

```
DSTG_MIN_SCHEDULING_INTERVAL = seconds
```

Notes:

1. You can only specify this keyword in the global configuration file; it will be ignored in local configuration files.
2. LoadLeveler ignores **DSTG_MIN_SCHEDULING_INTERVAL** when **DSTG_TIME=AT_SUBMIT**.

Default value: 900 seconds

DSTG_TIME

Specifies that either:

AT_SUBMIT
: LoadLeveler can schedule data staging steps any time after a job requiring data staging has been submitted.

JUST_IN_TIME
: LoadLeveler must schedule data staging job steps as close as possible to the application job steps that were submitted in the same job.

Syntax:

```
DSTG_TIME = AT_SUBMIT | JUST_IN_TIME
```

Note: You can only specify the **DSTG_TIME** keyword in the global configuration file. Any value specified for this keyword in local configuration files will be ignored.

Default value: AT_SUBMIT

ENFORCE_RESOURCE_MEMORY

Specifies whether the Workload Manager is configured to limit, as precisely as possible, the real memory usage of a WLM class. For this keyword to be valid, ConsumableMemory must be set through the **ENFORCE_RESOURCE_USAGE** keyword.

Syntax:

```
ENFORCE_RESOURCE_MEMORY = true | false
```

Default value: false

ENFORCE_RESOURCE_POLICY

Specifies what type of resource entitlements will be assigned to the Workload Manager classes. If the value specified is **shares**, it means a share value is assigned to the class based on the job step's requested resources (one unit of resource equals one share). This is the default policy. If the value specified is **soft**, it means a percentage value is assigned to the class based on the job step's requested resources and the total machine resources. This percentage can be exceeded if there is no contention for the resource. If the value specified is **hard**, it means a percentage value is assigned to the class based on the job step's requested resources and the total machine resources. This percentage cannot be exceeded regardless of the contention for the resource. This keyword is only valid for CPU and real memory with either shares or percent limits. If desired, this keyword can be used in the **LoadL_config.local** file to set up a different policy for each machine. The **ENFORCE_RESOURCE_USAGE** keyword must be set for this keyword to be valid.

Syntax:

```
ENFORCE_RESOURCE_POLICY = hard |soft | shares
```

Default value: shares

ENFORCE_RESOURCE_SUBMISSION

Indicates whether jobs submitted should be checked for the **resources** and **node_resources** keywords. If the value specified is **true**, LoadLeveler will check all jobs at submission time for the **resources** and **node_resources** keywords. The job command file **resources** and **node_resources** keywords combined need to have at least the resources specified in the **ENFORCE_RESOURCE_USAGE** keyword in order for the job to be submitted successfully. When **RSET_MCM_AFFINITY** is enabled, the **task_affinity** or **parallel_threads** keyword can be used instead of the **resources** and **node_resources** keywords when the resource being enforced is **ConsumableCpus**.

If the value specified is **false**, no checking will be done and jobs submitted without the **resources** or **node_resources** keywords will not have resources enforced. In this instance, those jobs might interfere with other jobs whose resources are enforced.

Syntax:

```
ENFORCE_RESOURCE_SUBMISSION = true | false
```

Default value: false

ENFORCE_RESOURCE_USAGE
Specifies whether the Workload Manager is used to enforce CPU and memory resources. This keyword accepts either a value of **deactivate** or a list of one or more of the following predefined resources:

- **ConsumableCpus**
- **ConsumableMemory**
- **ConsumableVirtualMemory**
- **ConsumableLargePageMemory**

Either memory or CPUs or both can be enforced but the resources must also be specified on the **SCHEDULE_BY_RESOURCES** keyword. If **deactivate** is specified, LoadLeveler will deactivate Workload Manager on all the nodes in the LoadLeveler cluster.

Syntax:

```
ENFORCE_RESOURCE_USAGE = name name ... name | deactivate
```

EXECUTE
Specifies the local directory to store the executables of jobs submitted by other machines.

Syntax:

```
EXECUTE = local directory/execute
```

Default value: `$(tilde)/execute`

EXT_ENERGY_POLICY_PROGRAM
Specifies a user-supplied executable to be run by Startd to determine the CPU frequency to use when running the job. For a job that has the energy function enabled, LoadLeveler will pass the job policy tag **LL_ENERGY_TAG_NAME** environment variable to the program. The output of the program is the frequency value the job will use.

When this keyword is enabled, LoadLeveler will ignore the **MAX_PERF_DECREASE_ALLOWED**, **ENERGY_SAVING_REQ**, and **ADJUST_WALL_CLOCK_LIMIT** keywords that are defined in the job command file. To accommodate the longer running time at a lower frequency, ensure the **WALL_CLOCK_LIMIT** is set high enough or change it by using the **llmodify** command.

Syntax:

```
EXT_ENERGY_POLICY_PROGRAM = full_path_to_executable
```

where:

full_path_to_executable
Is a user-supplied program that calculates the frequency the job will use.

Default value: NULL

FAILOVER_HEARTBEAT_INTERVAL
Specifies the amount of time, in seconds, that defines how frequently the primary and alternate central manager, resource manager, or region manager communicate with each other.

Syntax:

```
FAILOVER_HEARTBEAT_INTERVAL = seconds
```

Default value: The default value is 300 seconds or 5 minutes.

FAILOVER_HEARTBEAT_RETRIES

Specifies the number of heartbeat intervals that an alternate manager will wait before declaring that the primary central manager, resource manager, or region manager is not operating.

Syntax:

```
FAILOVER_HEARTBEAT_RETRIES = number
```

Default value: The default value is 6.

FAIR_SHARE_INTERVAL

Specifies, in units of hours, the time interval it takes for resource usage in fair share scheduling to decay to 5% of its initial value. Historic fair share data collected before the most recent time interval of this length will have little impact on fair share scheduling.

Syntax:

```
FAIR_SHARE_INTERVAL = hours
```

Default value: The default value is 168 hours (one week). If a negative value or 0 is specified, the default value is used.

FAIR_SHARE_TOTAL_SHARES

Specifies the total number of shares that the cluster CPU or Blue Gene resources are divided into. If this value is less than or equal to 0, fair share scheduling is turned off.

Syntax:

```
FAIR_SHARE_TOTAL_SHARES = shares
```

Default value: The default value is 0.

FEATURE

Specifies an optional characteristic to use to match jobs with machines. You can specify unique characteristics for any machine using this keyword. When evaluating job submissions, LoadLeveler compares any required features specified in the job command file to those specified using this keyword. You can have a maximum of 1024 characters in the feature statement.

Syntax:

```
Feature = {"string" ...}
```

Default value: No default value is set.

Example: If a machine has licenses for installed products ABC and XYZ in the local configuration file, you can enter the following:

```
Feature = {"abc" "xyz"}
```

When submitting a job that requires both of these products, you should enter the following in your job command file:

```
requirements = (Feature == "abc") && (Feature == "xyz")
```

Note: One optional way to run dynamic simultaneous multithreading (SMT) is to define a feature on all machines. SMT is only supported on POWER7 processor-based systems.

Example: When submitting a job that requires the SMT function, first specify **smt = yes** in the job command file (or select a class which had **smt = yes** defined). Next, specify **job_type = parallel** and **node_usage = not_shared** and last, enter the following in the job command file:

```
requirements = (Feature == "smt")
```

Note: This keyword is deprecated in the configuration file and has been moved to the administration file.

FLOATING_RESOURCES

Specifies which consumable resources are available collectively on all of the machines in the LoadLeveler cluster. The count for each resource must be an integer greater than or equal to zero, and each resource can only be specified once in the list. Any resource specified for this keyword that is not already listed in the **SCHEDULE_BY_RESOURCES** keyword will not affect job scheduling. If a resource is specified incorrectly with the **FLOATING_RESOURCES** keyword, then that resource will be ignored. All other correctly specified resources will be accepted. **ConsumableCpus**, **ConsumableMemory**, **ConsumableVirtualMemory**, and **ConsumableLargePageMemory** may not be specified as floating resources.

Syntax:

```
FLOATING_RESOURCES = name(count) name(count) ... name(count)
```

Default value: No default value is set.

FS_INTERVAL

Defines the number of minutes used as the interval for checking free file system space or inodes. If your file system receives many log messages or copies large executables to the LoadLeveler spool, the file system will fill up quicker and you should perform file size checking more frequently by setting the interval to a smaller value. LoadLeveler will not check the file system if the value of FS_INTERVAL is:

- Set to zero
- Set to a negative integer

Syntax:

```
FS_INTERVAL = minutes
```

Default value: If FS_INTERVAL is not specified but any of the other file-system keywords (FS_NOTIFY, FS_SUSPEND, FS_TERMINATE, INODE_NOTIFY, INODE_SUSPEND, INODE_TERMINATE) are specified, the FS_INTERVAL value will default to 5 and the file system will be checked. If no file-system or inode keywords are set, LoadLeveler does not monitor file systems at all.

For more information related to using this keyword, see "Setting up file system monitoring" on page 58.

FS_NOTIFY

Defines the lower and upper amounts, in bytes, of free file-system space at which LoadLeveler is to notify the administrator:

- If the amount of free space becomes less than the lower threshold value, LoadLeveler sends a mail message to the administrator indicating that logging problems may occur.
- When the amount of free space becomes greater than the upper threshold value, LoadLeveler sends a mail message to the administrator indicating that problem has been resolved.

Syntax:

```
FS_NOTIFY = lower threshold, upper threshold
```

Specify space in bytes with the unit B. A metric prefix such as K, M, or G may precede the B. The valid range for both the lower and upper thresholds are -1B and all positive integers. If the value is set to -1, the transition across the threshold is not checked.

Default value: In bytes: 1KB, -1B

For more information related to using this keyword, see "Setting up file system monitoring" on page 58.

FS_SUSPEND

Defines the lower and upper amounts, in bytes, of free file system space at which LoadLeveler drains and resumes the Schedd and startd daemons running on a node.

- If the amount of free space becomes less than the lower threshold value, then LoadLeveler drains the Schedd and the startd daemons if they are running on a node. When this happens, logging is turned off and mail notification is sent to the administrator.
- When the amount of free space becomes greater than the upper threshold value, LoadLeveler signals the Schedd and the startd daemons to resume. When this happens, logging is turned on and mail notification is sent to the administrator.

Syntax:

```
FS_SUSPEND = lower threshold, upper threshold
```

Specify space in bytes with the unit B. A metric prefix such as K, M, or G may precede the B. The valid range for both the lower and upper thresholds are -1B and all positive integers. If the value is set to -1, the transition across the threshold is not checked.

Default value: In bytes: -1B, -1B

For more information related to using this keyword, see "Setting up file system monitoring" on page 58.

FS_TERMINATE

Defines the lower and upper amounts, in bytes, of free file system space at which LoadLeveler is terminated. This keyword sends the SIGTERM signal to the Master daemon which then terminates all LoadLeveler daemons running on the node.

- If the amount of free space becomes less than the lower threshold value, all LoadLeveler daemons are terminated.
- An upper threshold value is required for this keyword. However, since LoadLeveler has been terminated at the lower threshold, no action occurs.

Syntax:

```
FS_TERMINATE = lower threshold, upper threshold
```

Specify space in bytes with the unit B. A metric prefix such as K, M, or G may precede the B. The valid range for the lower threshold is -1B and all positive integers. If the value is set to -1, the transition across the threshold is not checked.

Default value: In bytes: -1B, -1B

For more information related to using this keyword, see "Setting up file system monitoring" on page 58.

GLOBAL_HISTORY

Identifies the directory that will contain the global history files produced by **llacctmrg** command when no directory is specified as a command argument.

Syntax:

```
GLOBAL_HISTORY = directory
```

Default value: The default value is **$(SPOOL)** (the local spool directory).

For more information related to using this keyword, see "Collecting the accounting information and storing it into files" on page 69.

HISTORY

Defines the path name where a file containing the history of local LoadLeveler jobs is kept.

Syntax:

```
HISTORY = directory
```

Default value: $(SPOOL)/history

For more information related to using this keyword, see "Collecting the accounting information and storing it into files" on page 69.

HISTORY_PERMISSION

Specifies the owner, group, and world permissions of the history file associated with a **LoadL_schedd** daemon.

Syntax:

```
HISTORY_PERMISSION = permissions | rw-rw----
```

permissions must be a string with a length of nine characters and consisting of the characters, **r**, **w**, **x**, or **-**.

Default value: The default settings are 660 (**rw-rw----**). **LoadL_schedd** will use the default setting if the specified permission are less than **rw-------**.

Example: A specification such as `HISTORY_PERMISSION = rw-rw-r--` will result in permission settings of 664.

INODE_NOTIFY

Defines the lower and upper amounts, in inodes, of free file-system inodes at which LoadLeveler is to notify the administrator:

- If the number of free inodes becomes less than the lower threshold value, LoadLeveler sends a mail message to the administrator indicating that logging problems may occur.
- When the number of free inodes becomes greater than the upper threshold value, LoadLeveler sends a mail message to the administrator indicating that problem has been resolved.

Syntax:

```
INODE_NOTIFY = lower threshold, upper threshold
```

The valid range for both the lower and upper thresholds are -1 and all positive integers. If the value is set to -1, the transition across the threshold is not checked.

Default value: In inodes: 1000, -1

For more information related to using this keyword, see "Setting up file system monitoring" on page 58.

INODE_SUSPEND

Defines the lower and upper amounts, in inodes, of free file system inodes at which LoadLeveler drains and resumes the Schedd and startd daemons running on a node.

- If the number of free inodes becomes less than the lower threshold value, then LoadLeveler drains the Schedd and the startd daemons if they are

running on a node. When this happens, logging is turned off and mail notification is sent to the administrator.
- When the number of free inodes becomes greater than the upper threshold value, LoadLeveler signals the Schedd and the startd daemons to resume. When this happens, logging is turned on and mail notification is sent to the administrator.

Syntax:

```
INODE_SUSPEND = lower threshold, upper threshold
```

The valid range for both the lower and upper thresholds are -1 and all positive integers. If the value is set to -1, the transition across the threshold is not checked.

Default value: In inodes: -1, -1

For more information related to using this keyword, see "Setting up file system monitoring" on page 58.

INODE_TERMINATE

Defines the lower and upper amounts, in inodes, of free file system inodes at which LoadLeveler is terminated. This keyword sends the SIGTERM signal to the Master daemon which then terminates all LoadLeveler daemons running on the node.
- If the number of free inodes becomes less than the lower threshold value, all LoadLeveler daemons are terminated.
- An upper threshold value is required for this keyword. However, since LoadLeveler has been terminated at the lower threshold, no action occurs.

Syntax:

```
INODE_TERMINATE = lower threshold, upper threshold
```

The valid range for the lower threshold is -1 and all positive integers. If the value is set to -1, the transition across the threshold is not checked.

Default value: In inodes: -1, -1

For more information related to using this keyword, see "Setting up file system monitoring" on page 58.

JOB_ACCT_Q_POLICY

Specifies the amount of time, in seconds, that determines how often the startd daemon updates the Schedd daemon with accounting data of running jobs. This controls the accuracy of the **llq -x** command.

Syntax:

```
JOB_ACCT_Q_POLICY = number
```

Default value: The default is 300.

For more information related to using this keyword, see "Gathering job accounting data" on page 65.

JOB_EPILOG

Path name of the epilog program.

Syntax:

```
JOB_EPILOG = program name
```

Default value: No default value is set.

For more information related to using this keyword, see "Writing prolog and epilog programs" on page 80.

JOB_LIMIT_POLICY
Specifies the amount of time, in seconds, that LoadLeveler checks to see if **job_cpu_limit** has been exceeded. The smaller of **JOB_LIMIT_POLICY** and **JOB_ACCT_Q_POLICY** is used to control how often the **startd** daemon collects resource consumption data on running jobs, and how often the **job_cpu_limit** is checked.

Syntax:

```
JOB_LIMIT_POLICY = number
```

Default value: The default is 300.

JOB_PROLOG
Path name of the prolog program.

Syntax:

```
JOB_PROLOG = program name
```

Default value: No default value is set.

For more information related to using this keyword, see "Writing prolog and epilog programs" on page 80.

JOB_USER_EPILOG
Path name of the user epilog program.

Syntax:

```
JOB_USER_EPILOG = program name
```

Default value: No default value is set.

For more information related to using this keyword, see "Writing prolog and epilog programs" on page 80.

JOB_USER_PROLOG
Path name of the user prolog program.

Syntax:

```
JOB_USER_PROLOG = program name
```

Default value: No default value is set.

For more information related to using this keyword, see "Writing prolog and epilog programs" on page 80.

KBDD
Location of kbdd executable (**LoadL_kbdd**).

Syntax:

```
KBDD = directory
```

Default value: `$(BIN)/LoadL_kbdd`

KBDD_COREDUMP_DIR
Local directory for storing **LoadL_kbdd** daemon core dump files.

Syntax:

```
KBDD_COREDUMP_DIR = directory
```

Default value: The **/tmp** directory.

For more information related to using this keyword, see "Specifying file and directory locations" on page 49.

KILL

Determines whether or not vacated jobs should be sent the SIGKILL signal and replaced in the queue. It is used to remove a job that is taking too long to vacate.

Syntax:

```
KILL: expression that evaluates to T or F (true or false)
```

When **T**, vacated LoadLeveler jobs are removed from the machine with no attempt to take checkpoints.

For information about time-related variables that you may use for this keyword, see "Variables to use for setting times" on page 291.

LL_RSH_COMMAND

Specifies an administrator provided executable to be used by **llctl start** when starting LoadLeveler on remote machines in the administration file. The **LL_RSH_COMMAND** keyword is any executable that can be used as a substitute for **/usr/bin/rsh**. The **llctl start** command passes arguments to the executable specified by **LL_RSH_COMMAND** in the following format:

```
LL_RSH_COMMAND hostname -n llctl start options
```

Syntax:

```
LL_RSH_COMMAND = full_path_to_executable
```

Default value: /usr/bin/rsh. This keyword must specify the full path name to the executable provided. If no value is specified, LoadLeveler will use **/usr/bin/rsh** as the default when issuing a start. If an error occurred while locating the executable specified, an error message will be displayed.

Example: This example shows that using the secure shell (**/usr/bin/ssh**) is the preferred method for the **llctl start** command to communicate with remote nodes. Specify the following in the configuration file:

```
LL_RSH_COMMAND=/usr/bin/ssh
```

LOADL_ADMIN

Specifies a list of LoadLeveler administrators.

Syntax:

```
LOADL_ADMIN = list of user names
```

Where *list of user names* is a blank-delimited list of those individuals who will have administrative authority.

Default value: No default value is set, which means no one has administrator authority until this keyword is defined with one or more user names.

Example: To grant administrative authority to users `bob` and `mary`, enter the following in the configuration file:

```
LOADL_ADMIN = bob mary
```

For more information related to using this keyword, see "Defining LoadLeveler administrators" on page 45.

LOCAL_CONFIG

Specifies the path name of the optional local configuration file containing information specific to a node in the LoadLeveler network. This keyword is not used for the database configuration option.

Syntax:

```
LOCAL_CONFIG = directory
```

Default value: No default value is set.

Examples:

- If you are using a distributed file system like NFS, some examples are:

  ```
  LOCAL_CONFIG = $(tilde)/$(host).LoadL_config.local
  LOCAL_CONFIG = $(tilde)/LoadL_config.$(host).$(domain)
  LOCAL_CONFIG = $(tilde)/LoadL_config.local.$(hostname)
  ```

  See "LoadLeveler variables" on page 286 for information about the **tilde**, **host**, and **domain** variables.

- If you are using a local file system, an example is:

  ```
  LOCAL_CONFIG = /var/LoadL/LoadL_config.local
  ```

LOG

Defines the local directory to store log files. It is not necessary to keep all the log files created by the various LoadLeveler daemons and programs in one directory, but you will probably find it convenient to do so.

Syntax:

```
LOG = local directory/log
```

Default value: `$(tilde)/log`

LOG_MESSAGE_THRESHOLD

Specifies the maximum amount of memory, in bytes, for the message queue. Messages in the queue are waiting to be written to the log file. When the message logging thread cannot write messages to the log file as fast as they arrive, the memory consumed by the message queue can exceed the threshold. In this case, LoadLeveler will curtail logging by turning off all debug flags except **D_ALWAYS**, therefore, reducing the amount of logging that takes place. If the threshold is exceeded by the curtailed message queue, message logging is stopped. Special log messages are written to the log file, which indicate that some messages are missing. Mail is also sent to the administrator indicating that messages are missing. A value of -1 for this keyword will turn off the buffer threshold meaning that the threshold is unlimited.

Syntax:

```
LOG_MESSAGE_THRESHOLD = bytes
```

Default value: 20*1024*1024 (bytes)

MACHINE_AUTHENTICATE

Specifies whether machine validation is performed. When set to **true**, LoadLeveler only accepts connections from machines specified in the administration file. When set to **false**, LoadLeveler accepts connections from any machine.

When set to **true**, every communication between LoadLeveler processes will verify that the sending process is running on a machine which is identified via a machine stanza in the administration file. The validation is done by capturing the address of the sending machine when the **accept** function call is issued to accept a connection. The **gethostbyaddr** function is called to translate the address to a name, and the name is matched with the list derived from the administration file.

Syntax:

```
MACHINE_AUTHENTICATE = true | false
```

Default value: false

For more information related to using this keyword, see "Defining a LoadLeveler cluster" on page 45.

MACHINE_UPDATE_INTERVAL

Specifies the time, in seconds, during which machines must report to the central manager.

Syntax:

```
MACHINE_UPDATE_INTERVAL = number
```

Where *number* specifies the time period, in seconds, during which machines must report to the central manager. Machines that do not report in this number of seconds are considered *down*. *number* must be a numerical value and cannot be an arithmetic expression.

Default value: The default is 300 seconds.

For more information related to using this keyword, see "Setting negotiator characteristics and policies" on page 47.

MACHPRIO

Machine priority expression.

Syntax:

```
MACHPRIO = expression
```

You can use the following LoadLeveler variables in the **MACHPRIO** expression:

- **LoadAvg**
- **Connectivity**
- **Cpus**
- **Speed**
- **Memory**
- **VirtualMemory**
- **Disk**
- **CustomMetric**
- **MasterMachPriority**
- **ConsumableCpus**
- **ConsumableMemory**
- **ConsumableVirtualMemory**
- **ConsumableLargePageMemory**
- **PagesFreed**
- **PagesScanned**
- **FreeRealMemory**

For detailed descriptions of these variables, see "LoadLeveler variables" on page 286.

Default value: (0 - LoadAvg)

Examples:

- Example 1

  This example orders machines by the Berkeley one-minute load average.

  ```
  MACHPRIO : 0 - (LoadAvg)
  ```

  Therefore, if **LoadAvg** equals .7, this example would read:

  ```
  MACHPRIO : 0 - (.7)
  ```

  The **MACHPRIO** would evaluate to -.7.

- Example 2

  This example orders machines by the Berkeley one-minute load average normalized for machine speed:

  ```
  MACHPRIO : 0 - (1000 * (LoadAvg / (Cpus * Speed)))
  ```

  Therefore, if **LoadAvg** equals .7, **Cpus** equals 1, and **Speed** equals 2, this example would read:

  ```
  MACHPRIO : 0 - (1000 * (.7 / (1 * 2)))
  ```

  This example further evaluates to:

  ```
  MACHPRIO : 0 - (350)
  ```

  The **MACHPRIO** would evaluate to -350.

  Notice that if the speed of the machine were increased to 3, the equation would read:

  ```
  MACHPRIO : 0 - (1000 * (.7 / (1 * 3)))
  ```

  The **MACHPRIO** would evaluate to approximately -233. Therefore, as the speed of the machine increases, the **MACHPRIO** also increases.
- Example 3

  This example orders machines accounting for real memory and available swap space (remembering that Memory is in Mbytes and VirtualMemory is in Kbytes):

  ```
  MACHPRIO : 0 - (10000 * (LoadAvg / (Cpus * Speed))) +
  (10 * Memory) + (VirtualMemory / 1000)
  ```
- Example 4

  This example sets a relative machine priority based on the value of the **CUSTOM_METRIC** keyword.

  ```
  MACHPRIO : CustomMetric
  ```

  To do this, you must specify a value for the **CUSTOM_METRIC** keyword or the **CUSTOM_METRIC_COMMAND** keyword in either the **LoadL_config.local** file of a machine or in the global **LoadL_config** file. To assign the same relative priority to all machines, specify the **CUSTOM_METRIC** keyword in the global configuration file. For example:

  ```
  CUSTOM_METRIC = 5
  ```

  You can override this value for an individual machine by specifying a different value in that machine's **LoadL_config.local** file.
- Example 5

  This example gives master nodes the highest priority:

  ```
  MACHPRIO : (MasterMachPriority * 10000)
  ```
- Example 6

  This example gives nodes the with highest percentage of switch adapters with connectivity the highest priority:

  ```
  MACHPRIO : Connectivity
  ```

For more information related to using this keyword, see "Setting negotiator characteristics and policies" on page 47.

MAIL

Name of a local mail program used to override default mail notification.

Syntax:

```
MAIL = program name
```

Default value: No default value is set.

For more information related to using this keyword, see "Using your own mail program" on page 85.

MASTER
Location of the master executable (**LoadL_master**).

Syntax:

```
MASTER = file
```

Default value: `$(BIN)/LoadL_master`

For more information related to using this keyword, see "How LoadLeveler daemons process jobs" on page 8.

MASTER_COREDUMP_DIR
Local directory for storing **LoadL_master** core dump files.

Syntax:

```
MASTER_COREDUMP_DIR = directory
```

Default value: The **/tmp** directory.

For more information related to using this keyword, see "Specifying file and directory locations" on page 49.

MASTER_DGRAM_PORT
The port number used when connecting to the daemon.

Syntax:

```
MASTER_DGRAM_PORT = port number
```

Default value: The default is 9617.

For more information related to using this keyword, see "Defining network characteristics" on page 49.

MASTER_STREAM_PORT
Specifies the port number to be used when connecting to the daemon.

Syntax:

```
MASTER_STREAM_PORT = port number
```

Default value: The default is 9616.

For more information related to using this keyword, see "Defining network characteristics" on page 49.

MAX_CKPT_INTERVAL
The maximum number of seconds between checkpoints for running jobs.

Syntax:

```
MAX_CKPT_INTERVAL = number
```

where: *number* specifies the maximum period in seconds between checkpoints taken for running jobs.

Default value: 7200 (2 hours)

For more information related to using this keyword, see "LoadLeveler support for checkpointing jobs" on page 135.

MAX_JOB_REJECT
Determines the number of times a job is rejected before it is canceled or put in User Hold or System Hold status.

Syntax:

```
MAX_JOB_REJECT = number
```

number must be a numerical value and cannot be an arithmetic expression. **MAX_JOB_REJECT** may be set to unlimited rejects by specifying a value of –1.

Default value: The default value is 0, which indicates a rejected job will immediately be canceled or placed on hold.

For related information, see the **NEGOTIATOR_REJECT_DEFER** keyword.

MAX_RESERVATIONS

Specifies the maximum number of reservations that this LoadLeveler cluster can have. Only reservations in waiting and in use are counted toward this limit; LoadLeveler does not count reservations that have already ended or are in the process of being canceled.

Notes:

1. Having too many reservations in a LoadLeveler cluster can have performance impacts. Administrators should select a suitable value for this keyword.
2. A recurring reservation only counts as one reservation towards the **MAX_RESERVATIONS** limit regardless of the number of times that the reservation recurs.

Syntax:

```
MAX_RESERVATIONS = number
```

The value for this keyword can be 0 or a positive integer.

Default value: The default is 10.

MAX_STARTERS

Specifies the maximum number of tasks that can run simultaneously on a machine. In this case, a task can be a serial job step or a parallel task. **MAX_STARTERS** defines the number of initiators on the machine (the number of tasks that can be initiated from a **startd**).

Syntax:

```
MAX_STARTERS = number
```

Default value: If this keyword is not specified, the default is the number of elements in the **Class** statement.

For more information related to using this keyword, see "Specifying how many jobs a machine can run" on page 60.

Note: This keyword is deprecated in the configuration file and has been moved to the administration file.

MAX_TOP_DOGS

Specifies the maximum total number of top dogs that the central manager daemon will allocate. When scheduling jobs, after **MAX_TOP_DOGS** total top dogs have been allocated, no more will be considered.

Syntax:

```
MAX_TOP_DOGS = k | 1
```

where: *k* is a non-negative integer specifying the global maximum top dogs limit.

Default value: The default value is **1**.

For more information related to using this keyword, see "Using the BACKFILL scheduler" on page 114.

MIN_CKPT_INTERVAL
The minimum number of seconds between checkpoints for running jobs.

Syntax:

```
MIN_CKPT_INTERVAL = number
```

where: *number* specifies the initial period, in seconds, between checkpoints taken for running jobs.

Default value: 900 (15 minutes)

For more information related to using this keyword, see "LoadLeveler support for checkpointing jobs" on page 135.

NEGOTIATOR
Location of the negotiator executable (**LoadL_negotiator**).

Syntax:

```
NEGOTIATOR = file
```

Default value: `$(BIN)/LoadL_negotiator`

For more information related to using this keyword, see "How LoadLeveler daemons process jobs" on page 8.

NEGOTIATOR_COREDUMP_DIR
Local directory for storing **LoadL_negotiator** core dump files.

Syntax:

```
NEGOTIATOR_COREDUMP_DIR = directory
```

Default value: The **/tmp** directory.

For more information related to using this keyword, see "Specifying file and directory locations" on page 49.

NEGOTIATOR_CYCLE_DELAY
Specifies the minimum time, in seconds, the negotiator delays between periods when it attempts to schedule jobs. This time is used by the negotiator daemon to respond to queries, reorder job queues, collect information about changes in the states of jobs, and so on. Delaying the scheduling of jobs might improve the overall performance of the negotiator by preventing it from spending excessive time attempting to schedule jobs.

Syntax:

```
NEGOTIATOR_CYCLE_DELAY = number
```

number must be a numerical value and cannot be an arithmetic expression.

Default value: The default is 0 seconds

NEGOTIATOR_CYCLE_TIME_LIMIT
Specifies the maximum amount of time, in seconds, that LoadLeveler will allow the negotiator to spend in one cycle trying to schedule jobs. The negotiator cycle will end, after the specified number of seconds, even if there are additional jobs waiting for dispatch. Jobs waiting for dispatch will be considered at the next negotiator cycle. The **NEGOTIATOR_CYCLE_TIME_LIMIT** keyword applies only to the BACKFILL scheduler.

Syntax:

```
NEGOTIATOR_CYCLE_TIME_LIMIT = number
```

Where *number* must be a positive integer or zero and cannot be an arithmetic expression.

Default value: If the keyword value is not specified or a value of zero is used, the negotiator cycle will be unlimited.

NEGOTIATOR_INTERVAL
The time interval, in seconds, at which the negotiator daemon updates the status of jobs in the LoadLeveler cluster and negotiates with machines that are available to run jobs.

Syntax:

```
NEGOTIATOR_INTERVAL = number
```

Where *number* specifies the interval, in seconds, at which the negotiator daemon performs a "negotiation loop" during which it attempts to assign available machines to waiting jobs. A negotiation loop also occurs whenever job states or machine states change. *number* must be a numerical value and cannot be an arithmetic expression.

When this keyword is set to zero, the central manager's automatic scheduling activity is been disabled, and LoadLeveler will not attempt to schedule any jobs unless instructed to do so through the **llrunscheduler** command or **ll_run_scheduler** subroutine.

Default value: The default is 30 seconds.

For more information related to using this keyword, see "Controlling the central manager scheduling cycle" on page 76.

NEGOTIATOR_LOADAVG_INCREMENT
Specifies the value the negotiator adds to the startd machine's load average whenever a job in the Pending state is queued on that machine. This value is used to compensate for the increased load caused by starting another job.

Syntax:

```
NEGOTIATOR_LOADAVG_INCREMENT = number
```

number must be a numerical value and cannot be an arithmetic expression.

Default value: The default value is .5

NEGOTIATOR_PARALLEL_DEFER
Specifies the amount of time, in seconds, that defines how long a job stays out of the queue after it fails to get the correct number of processors. This keyword applies only to the default LoadLeveler scheduler. This keyword must be greater than the **NEGOTIATOR_INTERVAL**. value; if it is not, the default is used.

Syntax:

```
NEGOTIATOR_PARALLEL_DEFER = number
```

number must be a numerical value and cannot be an arithmetic expression.

Default value: The default is **NEGOTIATOR_INTERVAL** multiplied by 5.

NEGOTIATOR_PARALLEL_HOLD
Specifies the amount of time, in seconds, that defines how long a job is given to accumulate processors. This keyword applies only to the default

LoadLeveler scheduler. This keyword must be greater than the **NEGOTIATOR_INTERVAL** value; if it is not, the default is used.

Syntax:

```
NEGOTIATOR_PARALLEL_HOLD = number
```

number must be a numerical value and cannot be an arithmetic expression.

Default value: The default is **NEGOTIATOR_INTERVAL** multiplied by 5.

NEGOTIATOR_RECALCULATE_SYSPRIO_INTERVAL

Specifies the amount of time, in seconds, between calculation of the **SYSPRIO** values for waiting jobs. Recalculating the priority can be CPU-intensive; specifying low values for the **NEGOTIATOR_RECALCULATE_SYSPRIO_INTERVAL** keyword may lead to a heavy CPU load on the **negotiator** if a large number of jobs are running or waiting for resources. A value of 0 means the **SYSPRIO** values are not recalculated.

You can use this keyword to base the order in which jobs are run on the current number of running, queued, or total jobs for a user or a group.

Syntax:

```
NEGOTIATOR_RECALCULATE_SYSPRIO_INTERVAL = number
```

number must be a numerical value and cannot be an arithmetic expression.

Default value: The default is 120 seconds.

NEGOTIATOR_REJECT_DEFER

Specifies the amount of time in seconds the negotiator waits before it considers scheduling a job to a machine that recently rejected the job.

Syntax:

```
NEGOTIATOR_REJECT_DEFER = number
```

number must be a numerical value and cannot be an arithmetic expression.

Default value: The default is 120 seconds.

For related information, see the **MAX_JOB_REJECT** keyword.

NEGOTIATOR_REMOVE_COMPLETED

Specifies the amount of time, in seconds, that you want the negotiator to keep information regarding completed and removed jobs so that you can query this information using the **llq** command.

Syntax:

```
NEGOTIATOR_REMOVE_COMPLETED = number
```

number must be a numerical value and cannot be an arithmetic expression.

Default value: The default is 0 seconds.

NEGOTIATOR_RESCAN_QUEUE

specifies the amount of time in seconds that defines how long the negotiator waits to rescan the job queue for machines which have bypassed jobs which could not run due to conditions which may change over time. This keyword must be greater than the **NEGOTIATOR_INTERVAL** value; if it is not, the default is used.

Syntax:

```
NEGOTIATOR_RESCAN_QUEUE = number
```

number must be a numerical value and cannot be an arithmetic expression.

Default value: The default is 900 seconds.

NEGOTIATOR_STREAM_PORT
Specifies the port number used when connecting to the daemon.

Syntax:

```
NEGOTIATOR_STREAM_PORT = port number
```

Default value: The default is 9614.

For more information related to using this keyword, see "Defining network characteristics" on page 49.

OBITUARY_LOG_LENGTH
Specifies the number of lines from the end of the file that are appended to the mail message. The master daemon mails this log to the LoadLeveler administrators when one of the daemons dies.

Syntax:

```
OBITUARY_LOG_LENGTH = number
```

number must be a numerical value and cannot be an arithmetic expression.

Default value: The default is 25.

POLLING_FREQUENCY
Specifies the interval, in seconds, with which the startd daemon evaluates the load on the local machine and decides whether to suspend, resume, or abort jobs. This time is also the minimum interval at which the kbdd daemon reports keyboard or mouse activity to the startd daemon.

Syntax:

```
POLLING_FREQUENCY = number
```

number must be a numerical value and cannot be an arithmetic expression.

Default value: The default is 300.

POLLS_PER_UPDATE
Specifies how often, in **POLLING_FREQUENCY** intervals, startd daemon updates the central manager. Due to the communication overhead, it is impractical to do this with the frequency defined by the **POLLING_FREQUENCY** keyword. Therefore, the startd daemon only updates the central manager every *n*th (where *n* is the number specified for **POLLS_PER_UPDATE**) local update. Change **POLLS_PER_UPDATE** when changing the **POLLING_FREQUENCY**.

Syntax:

```
POLLS_PER_UPDATE = number
```

number must be a numerical value and cannot be an arithmetic expression.

Default value: The default is 1.

PRESTARTED_STARTERS
Specifies how many prestarted starter processes LoadLeveler will maintain on an execution node to manage jobs when they arrive. The startd daemon starts the number of starter processes specified by this keyword. You may specify this keyword in either the global or local configuration file.

Syntax:

PRESTARTED_STARTERS = *number*

number must be less than or equal to the value specified through the MAX_STARTERS keyword. If the value of PRESTARTED_STARTERS specified is greater then MAX_STARTERS, LoadLeveler records a warning message in the startd log and assigns PRESTARTED_STARTERS the same value as MAX_STARTERS.

If the value PRESTARTED_STARTERS is zero, no starter processes will be started before jobs arrive on the execution node.

Default value: The default is 1.

Note: This keyword is deprecated in the configuration file and has been moved to the administration file.

PREEMPT_CLASS

Defines the preemption rule for a job class.

Syntax: The following forms illustrate correct syntax.

PREEMPT_CLASS[*incoming_class***] = ALL[:***preempt_method***] { ***outgoing_class1*** [***outgoing_class2*** ...] }**

Using this form, ALL indicates that job steps of *incoming_class* have priority and will not share nodes with job steps of *outgoing_class1*, *outgoing_class2*, or other outgoing classes. If a job step of the *incoming_class* is to be started on a set of nodes, all job steps of *outgoing_class1*, *outgoing_class2*, or other outgoing classes running on those nodes will be preempted.

Note: The **ALL** preemption rule does not apply to Blue Gene jobs.

PREEMPT_CLASS[*incoming_class***] = ENOUGH[:***preempt_method***] { *outgoing_class1* [*outgoing_class2* ...] }**

Using this form, ENOUGH indicates that job steps of *incoming_class* will share nodes with job steps of *outgoing_class1*, *outgoing_class2*, or other outgoing classes if there are sufficient resources. If a job step of the *incoming_class* is to be started on a set of nodes, one or more job steps of *outgoing_class1*, *outgoing_class2*, or other outgoing classes running on those nodes may be preempted to get needed resources.

Combinations of these forms are also allowed.

Note:

1. The optional specification *preempt_method* indicates which method LoadLeveler is to use to preempt the jobs; this specification is valid only for the BACKFILL scheduler. Valid values for this specification in keyword syntax are the highlighted abbreviations in parentheses:
   - Remove (**rm**)
   - System hold (**sh**)
   - Suspend (**su**)
   - Vacate (**vc**)
   - User hold (**uh**)

   For more information about preemption methods, see "Steps for configuring a scheduler to preempt jobs" on page 126.
2. Using the "ALL" value in the PREEMPT_CLASS keyword places implied restrictions on when a job can start. See "Planning to preempt jobs" on page 124 for more information.

3. The incoming class is designated inside [ ] brackets.
4. Outgoing classes are designated inside { } curly braces.
5. The job classes on the right hand (outgoing) side of the statement must be different from incoming class, or it may be **allclasses**. If the outgoing side is defined as **allclasses** then all job classes are preemptable with the exception of the incoming class specified within brackets.
6. A class name or **allclasses** should not be in both the ALL list and the ENOUGH list. If you do so, the entire statement will be ignored. An example of this is:

   ```
   PREEMPT_CLASS[Class_A]=ALL{allclasses} ENOUGH {allclasses}
   ```

7. If you use **allclasses** as an outgoing (preemptable) class, then no other class names should be listed at the right hand side as the entire statement will be ignored. An example of this is:

   ```
   PREEMPT_CLASS[Class_A]=ALL{Class_B} ENOUGH {allclasses}
   ```

8. More than one ALL statement and more than one ENOUGH statement may appear at the right hand side. Multiple statements have a cumulative effect.
9. Each ALL or ENOUGH statement can have multiple class names inside the curly braces. However, a blank space delimiter is required between each class name.
10. Both the ALL and ENOUGH statements can include an optional specification indicating the method LoadLeveler will use to preempt the jobs. Valid values for this specification are listed in the description of the DEFAULT_PREEMPT_METHOD keyword. If a value is specified on the PREEMPT_CLASS ALL or ENOUGH statement, that value overrides the value set on the DEFAULT_PREEMPT_METHOD keyword, if any.
11. ALL and ENOUGH may be in mixed cases.
12. Spaces are allowed around the brackets and curly braces.
13. PREEMPT_CLASS [allclasses] will be ignored.

Default value: No default value is set.

Examples:

```
PREEMPT_CLASS[Class_B]=ALL{Class_E Class_D} ENOUGH {Class_C}
```
This indicates that all Class_E jobs and all Class_D jobs and enough Class_C jobs will be preempted to enable an incoming Class_B job to run.

```
PREEMPT_CLASS[Class_D]=ENOUGH:VC {Class_E}
```
This indicates that zero, one, or more Class_E jobs will be preempted using the vacate method to enable an incoming Class_D job to run.

PREEMPTION_SUPPORT

For the BACKFILL or API schedulers only, specifies the level of preemption support for a cluster.

Syntax:

```
PREEMPTION_SUPPORT= full | no_adapter | none
```

- When set to **full**, preemption is fully supported.
- When set to **no_adapter**, preemption is supported but the adapter resources are not released by preemption.
- When set to **none**, preemption is not supported, and preemption requests will be rejected.

Note:

1. If the value of this keyword is set to any value other than **none** for the default scheduler, LoadLeveler will not start.
2. For the BACKFILL or API scheduler, when this keyword is set to **full** or **no_adapter** and preemption by the suspend method is required, the configuration keyword **PROCESS_TRACKING** must be set to **true**.
3. The environment variable **MP_DEBUG_COMM_TIMEOUT=yes** must be set when the **no_adapter** option is used for preemption on the IBM Power 775 server.

Default value: The default value for all schedulers is **none**; if you want to enable preemption under these schedulers, you must set a value for this keyword.

PROCESS_TRACKING

Specifies whether or not LoadLeveler will cancel any processes (throughout the entire cluster), left behind when a job terminates.

Syntax:

```
PROCESS_TRACKING = TRUE | FALSE
```

When **TRUE** ensures that when a job is terminated, no processes created by the job will continue running.

Note: It is necessary to set this keyword to **true** to do preemption by the suspend method with the BACKFILL or API scheduler.

Default value: FALSE

PROCESS_TRACKING_EXTENSION

Specifies the directory containing the kernel module **LoadL_pt_ke** (AIX only).

Syntax:

```
PROCESS_TRACKING_EXTENSION = directory
```

Default value: The directory **$HOME/bin**

For more information related to using this keyword, see "Tracking job processes" on page 73.

PUBLISH_OBITUARIES

Specifies whether or not the master daemon sends mail to the administrator when any daemon it manages ends abnormally. When set to **true**, this keyword specifies that the master daemon sends mail to the administrators identified by **LOADL_ADMIN** keyword.

Syntax:

```
PUBLISH_OBITUARIES = true | false
```

Default value: true

RAS_MSG_FILE_DIR

Specifies the directory where the Reliability, Availability, and Serviceability (RAS) message files or RAS logs are stored. When a cluster uses file-based configuration, the manager daemons will write the RAS logs into this directory. When a cluster uses the database, this directory will be used as the temporary space when writing RAS messages into the database.

Syntax:

```
RAS_MSG_FILE_DIR = directory
```

Default value: $LOG

REGION_MGR
Specifies the location of the region manager executable (**LoadL_region_mgr**). This keyword is used by the resource manager component only.

Syntax:

```
REGION_MGR = file
```

Default value: `$(BIN)/LoadL_region_mgr`

REGION_MGR_COREDUMP_DIR
Local directory for storing **LoadL_region_mgr** core dump files. This keyword is used by the resource manager component only.

Syntax:

```
REGION_MGR_COREDUMP_DIR = directory
```

Default value: The **/tmp** directory.

For more information related to using this keyword, see "Specifying file and directory locations" on page 49.

REGION_MGR_DGRAM_PORT
Specifies the port number used when connecting to the daemon. This keyword is used by the resource manager component only.

Syntax:

```
REGION_MGR_DGRAM_PORT = port number
```

Default value: The default is 9682.

For more information related to using this keyword, see "Defining network characteristics" on page 49.

REGION_MGR_STREAM_PORT
Specifies the port number used when connecting to the daemon. This keyword is used by the resource manager component only.

Syntax:

```
REGION_MGR_STREAM_PORT = port number
```

Default value: The default is 9680.

For more information related to using this keyword, see "Defining network characteristics" on page 49.

REJECT_ON_RESTRICTED_LOGIN
Specifies whether the user's account status will be checked on every node where the job will be run by calling the AIX **loginrestrictions** function with the **S_DIST_CLNT** flag.

Restriction: Login restriction checking is ignored by LoadLeveler for Linux.

Login restriction checking includes:
- Does the account still exist?
- Is the account locked?
- Has the account expired?
- Do failed login attempts exceed the limit for this account?
- Is login disabled via **/etc/nologin**?

If the AIX **loginrestrictions** function indicates a failure then the user's job will be rejected and will be processed according to the LoadLeveler configuration parameters **MAX_JOB_REJECT** and **ACTION_ON_MAX_REJECT**.

Syntax:

```
REJECT_ON_RESTRICTED_LOGIN = true | false
```

Default value: false

RELEASEDIR
Defines the directory where all the LoadLeveler software resides.

Syntax:

```
RELEASEDIR = release directory
```

Default value: $(RELEASEDIR)

RESERVATION_CAN_BE_EXCEEDED
Specifies whether LoadLeveler will schedule job steps that are bound to a reservation when their end times (based on hard wall-clock limits) exceed the reservation end time.

Syntax:

```
RESERVATION_CAN_BE_EXCEEDED = true | false
```

When this keyword is set to false, LoadLeveler schedules only those job steps that will complete before the reservation ends. When set to true, LoadLeveler schedules job steps to run under a reservation even if their end times are expected to exceed the reservation end time. When the reservation ends, however, the reserved nodes no longer belong to the reservation, and so these nodes might not be available for the jobs to continue running. In this case, LoadLeveler might preempt the running jobs.

Note that this keyword setting does not change the actual end time of the reservation. It only affects how LoadLeveler manages job steps whose end times exceed the end time of the reservation.

Default value: true

RESERVATION_HISTORY
Defines the name of a file that is to contain the local history of reservations.

Syntax:

```
RESERVATION_HISTORY = file name
```

LoadLeveler appends a single line to the reservation history file for each completed occurrence of each reservation. For an example, see "Collecting accounting data for reservations" on page 67.

Default value: $(SPOOL)/reservation_history

RESERVATION_MIN_ADVANCE_TIME
Specifies the minimum time, in minutes, between the time at which a reservation is created and the time at which the reservation is to start.

Syntax:

```
RESERVATION_MIN_ADVANCE_TIME = number of minutes
```

By default, the earliest time at which a reservation may start is the current time plus the value set for the **RESERVATION_SETUP_TIME** keyword.

Default value: 0 (zero)

RESERVATION_PRIORITY
Specifies whether LoadLeveler administrators may reserve nodes on which running jobs are expected to end after the reservation start time. This keyword value applies only for LoadLeveler administrators; other reservation owners do not have this capability.

Syntax:

```
RESERVATION_PRIORITY = NONE | HIGH
```

When you set this keyword to HIGH, before activating the reservation, LoadLeveler preempts the job steps running on the reserved nodes (Blue Gene job steps are handled the same way). The only exceptions are non-preemptable jobs; LoadLeveler will not preempt those jobs because of any reservations.

Default value: NONE

RESERVATION_SETUP_TIME
Specifies how much time, in seconds, that LoadLeveler may use to prepare for a reservation before it is to start. The tasks that LoadLeveler performs during this time include checking and reporting node conditions, and preempting job steps still running on the reserved nodes.

For a given reservation, LoadLeveler uses the **RESERVATION_SETUP_TIME** keyword value that is set at the time that the reservation is created, not whatever value might be set when the reservation starts. If the start time of the reservation is modified, however, LoadLeveler uses the **RESERVATION_SETUP_TIME** keyword value that is set at the time of the modification.

Syntax:

```
RESERVATION_SETUP_TIME = number of seconds
```

Default value: 60

RESOURCE_MGR
Specifies the location of the resource manager daemon executable (LoadL_resource_mgr). This keyword is used by the resource manager component only.

Syntax:

```
RESOURCE_MGR = file
```

Default value: $(BIN)/LoadL_resource_mgr

RESOURCE_MGR_COREDUMP_DIR
Specifies the local directory for storing LoadL_resource_mgr core dump files. This keyword is used by the resource manager component only.

Syntax:

```
RESOURCE_MGR_COREDUMP_DIR = directory
```

Default value: The /tmp directory

RESOURCE_MGR_DGRAM_PORT
Specifies the port number used when connecting to the daemon. This keyword is used by the resource manager component only.

Syntax:

```
RESOURCE_MGR_DGRAM_PORT = port_number
```

Default value: The default is 9619

RESOURCE_MGR_LIST
Specifies the machines where the primary and alternate resource manager daemons run. If no machines are specified, the central manager list will be used for the resource manager list. This keyword is used by the resource manager component only.

Syntax:

```
RESOURCE_MGR_LIST = primary_resource_manager_machine \
                    [alternate_resource_manager_machine_list]
```

where:

primary_resource_manager_machine is the hostname of the machine on which the primary resource manager daemon will run.

[alternate_resource_manager_machine_list] is a blank-delimited list of hostnames for the alternate resource manager daemons.

RESOURCE_MGR_STREAM_PORT

Specifies the port number used when connecting to the daemon. This keyword is used by the resource manager component only.

Syntax:

```
RESOURCE_MGR_STREAM_PORT = port_number
```

Default value: The default value is 9618.

RESTARTS_PER_HOUR

Specifies how many times the master daemon attempts to restart a daemon that dies abnormally. Because one or more of the daemons may be unable to run due to a permanent error, the master only attempts **$(RESTARTS_PER_HOUR)** restarts within a 60 minute period. Failing that, it sends mail to the administrators identified by the **LOADL_ADMIN** keyword and exits.

Syntax:

```
RESTARTS_PER_HOUR = number
```

number must be a numerical value and cannot be an arithmetic expression.

Default value: The default is 12.

RESUME_ON_SWITCH_TABLE_ERROR_CLEAR

Specifies whether or not the **startd** that was drained when the switch table failed to unload will automatically resume once the unload errors are cleared. The unload error is considered cleared after LoadLeveler can successfully unload the switch table. For this keyword to work, the **DRAIN_ON_SWITCH_TABLE_ERROR** option in the configuration file must be turned on and not disabled. Flushing, suspending, or draining of a **startd** manually or automatically will disable this option until the **startd** is manually resumed.

Syntax:

```
RESUME_ON_SWITCH_TABLE_ERROR_CLEAR = true | false
```

Default value: false

RSET_SUPPORT

Indicates the level of RSet support present on a machine.

Syntax:

```
RSET_SUPPORT = option
```

The available options are:

RSET_MCM_AFFINITY
: Indicates that the machine can run jobs requesting MCM (memory or adapter) and processor affinity.

RSET_NONE
: Indicates that LoadLeveler RSet support is not available on the machine.

RSET_USER_DEFINED
Indicates that the machine can be used for jobs with a user-created RSet in their job command file.

Note: The **RSET_USER_DEFINED** option is not supported on Linux on x86 or Linux on POWER systems.

Default value: RSET_NONE

SAVELOGS
Specifies the directory in which log files are archived.

Syntax:

```
SAVELOGS = directory
```

Where *directory* is the directory in which log files will be archived.

Default value: No default value is set.

For more information related to using this keyword, see "Configuring recording activity and log files" on page 52.

SAVELOGS_COMPRESS_PROGRAM
Compresses logs after they are copied to the **SAVELOGS** directory. If not specified, **SAVELOGS** are copied, but are not compressed.

Syntax:

```
SAVELOGS_COMPRESS_PROGRAM = program
```

Where *program* is a specific executable program. It can be a system-provided facility (such as, **/bin/gzip**) or an administrator-provided executable program. The value must be a full path name and can contain command-line arguments. LoadLeveler will call the program as: *program filename*.

Default value: If blank, the logs are not compressed.

Example: In this example, LoadLeveler will run the **gzip -f** command. The log file in **SAVELOGS** will be compressed after it is copied to **SAVELOGS**. If the *program* cannot be found or is not executable, LoadLeveler will log the error and *SAVELOGS* will remain uncompressed.

```
SAVELOGS_COMPRESS_PROGRAM = /bin/gzip -f
```

SCHEDD
Location of the Schedd executable (**LoadL_schedd**).

Syntax:

```
SCHEDD = file
```

Default value: `$(BIN)/LoadL_schedd`

For more information related to using this keyword, see "How LoadLeveler daemons process jobs" on page 8.

SCHEDD_COREDUMP_DIR
Specifies the local directory for storing **LoadL_schedd** core dump files.

Syntax:

```
SCHEDD_COREDUMP_DIR = directory
```

Default value: The **/tmp** directory.

For more information related to using this keyword, see "Specifying file and directory locations" on page 49.

SCHEDD_INTERVAL

Specifies the interval, in seconds, at which the Schedd daemon checks the local job queue and updates the negotiator daemon.

Syntax:

```
SCHEDD_INTERVAL = number
```

number must be a numerical value and cannot be an arithmetic expression.

Default value: The default is 60 seconds.

SCHEDD_RUNS_HERE

Specifies whether the Schedd daemon runs on the host. If you do not want to run the Schedd daemon, specify **false**.

This keyword does not designate a machine as a public scheduling machine. Unless configured as a public scheduling machine, a machine configured to run the Schedd daemon will only accept job submissions from the same machine running the Schedd daemon. A public scheduling machine accepts job submissions from other machines in the LoadLeveler cluster. To configure a machine as a public scheduling machine, see the **schedd_host** keyword description in "Administration keyword descriptions" on page 298.

Syntax:

```
SCHEDD_RUNS_HERE = true | false
```

Default value: true

Note: This keyword is deprecated in the configuration file and has been moved to the administration file.

SCHEDD_SUBMIT_AFFINITY

Specifies whether job submissions are directed to a locally running Schedd daemon. When the keyword is set to **true**, job submissions are directed to a Schedd daemon running on the same machine where the submission takes place, provided there is a Schedd daemon running on that machine. In this case the submission is said to have "affinity" for the local Schedd daemon. If there is no Schedd daemon running on the machine where the submission takes place, or if this keyword is set to **false**, the job submission will only be directed to a Schedd daemon serving as a public scheduling machine. In this case, if there are no public scheduling machines configured the job cannot be submitted. A public scheduling machine accepts job submissions from other machines in the LoadLeveler cluster. To configure a machine as a public scheduling machine, see the **schedd_host** keyword description in "Administration keyword descriptions" on page 298.

Installations with a large number of nodes should consider setting this keyword to **false** to more evenly distribute dispatching of jobs among the Schedd daemons. For more information, see "Scaling considerations" on page 411.

Syntax:

```
SCHEDD_SUBMIT_AFFINITY = true | false
```

Default value: true

SCHEDD_STATUS_PORT

Specifies the port number used when connecting to the daemon.

Syntax:

```
SCHEDD_STATUS_PORT = port number
```

Default value: The default is 9606.

For more information related to using this keyword, see "Defining network characteristics" on page 49.

SCHEDD_STREAM_PORT
Specifies the port number used when connecting to the daemon.

Syntax:

```
SCHEDD_STREAM_PORT = port number
```

Default value: The default is 9605.

For more information related to using this keyword, see "Defining network characteristics" on page 49.

SCHEDULE_BY_RESOURCES
Specifies which consumable resources are considered by the LoadLeveler schedulers. Each consumable resource name may be an administrator-defined alphanumeric string, or may be one of the following predefined resources:

- **ConsumableCpus**
- **ConsumableMemory**
- **ConsumableVirtualMemory**
- **ConsumableLargePageMemory**
- **RDMA**
- **CollectiveGroups**

Each string may only appear in the list once. These resources are either floating resources, or machine resources. If any resource is specified incorrectly with the **SCHEDULE_BY_RESOURCES** keyword, then all scheduling resources will be ignored.

When using IBM Host Fabric Interface (HFI) switch adapters, the predefined resource **CollectiveGroups** must be in the **SCHEDULE_BY_RESOURCES** list.

Syntax:

```
SCHEDULE_BY_RESOURCES = name name ... name
```

Default value: No default value is set.

SCHEDULER_TYPE
Specifies the LoadLeveler scheduling algorithm:

LL_DEFAULT
Specifies the default LoadLeveler scheduling algorithm. If SCHEDULER_TYPE has not been defined, LoadLeveler will use the default scheduler (LL_DEFAULT).

BACKFILL
Specifies the LoadLeveler BACKFILL scheduler. When you specify this keyword, you should use only the default settings for the **START** expression and the other job control expressions described in "Managing job status through control expressions" on page 72.

API Specifies that you will use an external scheduler. External schedulers communicate to LoadLeveler through the job control API. For more information on setting an external scheduler, see "Using an external scheduler" on page 119.

Syntax:

```
SCHEDULER_TYPE = LL_DEFAULT | BACKFILL | API
```

Default value: LL_DEFAULT

Note:

1. If a scheduler type is not set, LoadLeveler will start, but it will use the default scheduler.
2. If you have set **SCHEDULER_TYPE** with an option that is not valid, LoadLeveler will not start.
3. If you change the scheduler option specified by **SCHEDULER_TYPE**, you must stop and restart LoadLeveler using **llctl** or recycle using **llctl**.

For more information related to using this keyword, see "Defining a LoadLeveler cluster" on page 45.

SEC_ADMIN_GROUP

When security services are enabled, this keyword points to the name of the UNIX group that contains the local identities of the LoadLeveler administrators.

Restriction: CtSec security is not supported on LoadLeveler for Linux.

Syntax:

```
SEC_ADMIN_GROUP = name of lladmin group
```

Default value: No default value is set.

For more information related to using this keyword, see "Configuring LoadLeveler to use cluster security services" on page 61.

SEC_ENABLEMENT

Specifies the security mechanism to be used.

Restriction: Do not set this keyword to CtSec in the configuration file for a Linux machine. CtSec security is not supported on LoadLeveler for Linux.

Syntax:

```
SEC_ENABLEMENT = COMPAT | CTSEC
```

Default value: No default value is set.

SEC_SERVICES_GROUP

When security services are enabled, this keyword specifies the name of the LoadLeveler services group.

Restriction: CtSec security is not supported on LoadLeveler for Linux.

Syntax:

```
SEC_SERVICES_GROUP=group name
```

Where *group name* defines the identities of the LoadLeveler daemons.

Default value: No default value is set.

SEC_IMPOSED_MECHS

Specifies a blank-delimited list of LoadLeveler's permitted security mechanisms when Cluster Security (CtSec) services are enabled.

Restriction: CtSec security is not supported on LoadLeveler for Linux.

Syntax: Specify a blank delimited list containing combinations of the following values:

none If this is the only value specified, then users *will* run unauthenticated and, if authorization is necessary, the job will fail. If this is not the only value specified, then users *may* run unauthenticated and, if authorization is necessary, the job will fail.

unix If this is the only value specified, then UNIX host-based authentication will be used; otherwise, other mechanisms may be used.

Default value: No default value is set.

Example:

```
SEC_IMPOSED_MECHS = none unix
```

SPOOL

Defines the local directory where LoadLeveler keeps the local job queue and checkpoint files.

Syntax:

SPOOL = *local directory*/**spool**

Default value: $(tilde)/spool

SSHD_PORTS

Port numbers used by sshd daemons started by LoadLeveler for interactive jobs.

Syntax:

SSHD_PORTS = *port number*

Where the *port number* list can be a blank delimited list of port numbers or a range of port number specified by two numbers separated by a '-'.

Default value: 9620-9629.

Ports defined for this purpose can only be specified in the LoadLeveler configuration. LoadLeveler does not look in **/etc/services** for these ports.

STALE_ENERGY_TAG_CLEANUP

Removes the energy tag from the database automatically when an energy tag has not been referenced in the last *the_number_of_days*. LoadLeveler will not remove the energy tag by default.

Syntax:

STALE_ENERGY_TAG_CLEANUP = *the_number_of_days*

where:

the_number_of_days
Indicates the days that the energy tag has not been used.

Default value: -1.

Example:

Remove energy tags that have not been used in 30 days from the database:

```
STALE_ENERGY_TAG_CLEANUP = 30
```

START

Determines whether a machine can run a LoadLeveler job.

Syntax:

START: *expression that evaluates to T or F (true or false)*

When the expression evaluates to **T**, LoadLeveler considers dispatching a job to the machine. When you use a START expression that is based on the CPU load average, the negotiator may evaluate the expression as **F** even though the load average indicates the machine is Idle. This is because the negotiator adds

a compensating factor to the startd machine's load average every time the negotiator assigns a job. For more information, see the NEGOTIATOR_INTERVAL keyword.

Default value: No default value is set, which means that no jobs will be started.

For information about time-related variables that you may use for this keyword, see "Variables to use for setting times" on page 291.

START_CLASS

Specifies the rule for starting a job of the *incoming_class*. The START_CLASS rule is applied whenever the BACKFILL scheduler decides whether a job step of the *incoming_class* should start or not.

Syntax:

```
START_CLASS[incoming_class] = (start_class_expression) [ && (start_class_expression) ...]
```

Where *start_class_expression* takes the form:

run_class < number_of_tasks

Which indicates that a job step of the *incoming_class* is only allowed to run on a node when the number of tasks of *run_class* running on that node is less than *number_of_tasks*.

Notes:

1. START_CLASS [**allclasses**] will be ignored.
2. The job class specified by *run_class* may be the same as or different from the class specified by *incoming_class*.
3. You can also define *run_class* as **allclasses**. If you do, the total number of all job tasks running on that node cannot exceed the value specified by *number_of_tasks*.
4. A class name or **allclasses** should not appear twice on the right-hand side of the keyword statement. However, you can use other class names with **allclasses** on the right hand side of the statement.
5. If there is more than one *start_class_expression*, you must use && between adjacent *start_class_expression*s.
6. Both the START keyword and the START_CLASS keyword have to be true before a new job can start.
7. Parenthesis ( ) are optional around *start_class_expression*.

For information related to using this keyword, see "Planning to preempt jobs" on page 124.

Default value: No default value is set.

Examples:

START_CLASS[Class_A] = (Class_A < 1)

This statement indicates that a Class_A job can only start on nodes that do not have any Class_A jobs running.

START_CLASS[Class_B] = allclasses < 5

This statement indicates that a Class_B job can only start on nodes with maximum 4 tasks running.

START_DAEMONS

Specifies whether to start the LoadLeveler daemons on the node.

Syntax:

START_DAEMONS = **true** | false

Default value: true

When **true**, the daemons are started. In most cases, you will probably want to set this keyword to **true**. An example of why this keyword would be set to **false** is if you want to run the daemons on most of the machines in the cluster but some individual users with their own local configuration files do not want their machines to run the daemons. The individual users would modify their local configuration files and set this keyword to **false**. Because the global configuration file has the keyword set to **true**, their individual machines would still be able to participate in the LoadLeveler cluster.

Also, to define the machine as strictly a submit-only machine, set this keyword to **false**.

STARTD

Location of the startd executable (**LoadL_startd**).

Syntax:

STARTD = *file*

Default value: $(BIN)/LoadL_startd

For more information related to using this keyword, see "How LoadLeveler daemons process jobs" on page 8.

STARTD_COREDUMP_DIR

Local directory for storing **LoadL_startd** core dump files.

Syntax:

STARTD_COREDUMP_DIR = *directory*

Default value: The **/tmp** directory.

For more information related to using this keyword, see "Specifying file and directory locations" on page 49.

STARTD_DGRAM_PORT

Specifies the port number used when connecting to the daemon.

Syntax:

STARTD_DGRAM_PORT = *port number*

Default value: The default is 9615.

For more information related to using this keyword, see "Defining network characteristics" on page 49.

STARTD_RUNS_HERE = true | false

Specifies whether the startd daemon runs on the host. If you do not want to run the startd daemon, specify **false**.

Syntax:

STARTD_RUNS_HERE = **true** | false

Default value: true

Note: This keyword is deprecated in the configuration file and has been moved to the administration file.

STARTD_STREAM_PORT

Specifies the port number used when connecting to the daemon.

Syntax:

STARTD_STREAM_PORT = *port number*

Default value: The default is 9611.

For more information related to using this keyword, see "Defining network characteristics" on page 49.

STARTER
Location of the starter executable (**LoadL_starter**).

Syntax:

STARTER = *directory*

Default value: $(BIN)/LoadL_starter

For more information related to using this keyword, see "How LoadLeveler daemons process jobs" on page 8.

STARTER_COREDUMP_DIR
Local directory for storing **LoadL_starter** coredump files.

Syntax:

STARTER_COREDUMP_DIR = *directory*

Default value: The **/tmp** directory.

For more information related to using this keyword, see "Specifying file and directory locations" on page 49.

SUBMIT_FILTER
Specifies the program you want to run to filter a job script when the job is submitted.

Syntax:

SUBMIT_FILTER = *full_path_to_executable*

Where *full_path_to_executable* is called with the job command file as the standard input. The standard output is submitted to LoadLeveler. If the program returns with a nonzero exit code, the job submission is canceled. A submit filter can only make changes to LoadLeveler job command file keyword statements.

Default value: No default value is set.

Multicluster use: In a multicluster environment, if you specified a valid cluster list with either the **llsubmit -X** option or the **ll_cluster** API, then the **SUBMIT_FILTER** will instead be invoked with a modified job command file that contains a **cluster_list** keyword generated from either the **llsubmit -X** option or the **ll_cluster** API.

The modified job command file will contain an inserted # @ cluster_list = *cluster* statement just prior to the first # @ queue statement. This **cluster_list** statement takes precedence and overrides all previous specifications of any **cluster_list** statements from the original job command file.

Example: SUBMIT_FILTER in a multicluster environment

The following job command file, job.cmd, requests to be run remotely on cluster1:

```
#!/bin/sh
# @ cluster_list = cluster1
# @ error = job1.$(Host).$(Cluster).$(Process).err
# @ output = job1.$(Host).$(Cluster).$(Process).out
# @ queue
```

After issuing `llsubmit -X cluster2 job.cmd`, the modified job command file statements will be run on cluster2:

```
#!/bin/sh
# @ cluster_list = cluster1
# @ error = job1.$(Host).$(Cluster).$(Process).err
# @ output = job1.$(Host).$(Cluster).$(Process).out
# @ cluster_list = cluster2
# @ queue
```

For more information related to using this keyword, see "Filtering a job script" on page 79.

SUSPEND
Determines whether running jobs should be suspended.

Syntax:

`SUSPEND: *expression that evaluates to T or F (true or false)*`

When **T**, LoadLeveler temporarily suspends jobs currently running on the machine. Suspended LoadLeveler jobs will either be continued or vacated. This keyword is not supported for parallel jobs.

Default value: No default value is set.

For information about time-related variables that you may use for this keyword, see "Variables to use for setting times" on page 291.

SUSPEND_CONTROL
Specifies the action to take when LoadLeveler fails to suspend an idle machine.

Syntax:

`SUSPEND_CONTROL = noact | reset | shutdown`

The available actions are:

shutdown
Shuts the failed machine down.

reset
Reboots the failed machine.

noact
No action is taken. Keep the failed machine in the current state.

Default value: noact

SYSPRIO
System priority expression.

Syntax:

`SYSPRIO : *expression*`

You can use the following LoadLeveler variables to define the **SYSPRIO** expression:
- **ClassSysprio**
- **GroupQueuedJobs**
- **GroupRunningJobs**
- **GroupSysprio**
- **GroupTotalJobs**
- **GroupTotalShares**
- **GroupUsedBgShares**
- **GroupUsedShares**
- **JobIsBlueGene**

- **QDate**
- **UserHoldTime**
- **UserPrio**
- **UserQueuedJobs**
- **UserRunningJobs**
- **UserSysprio**
- **UserTotalJobs**
- **UserTotalShares**
- **UserUsedBgShares**
- **UserUsedShares**

For detailed descriptions of these variables, see "LoadLeveler variables" on page 286.

Default value: noact

Notes:

1. The **SYSPRIO** keyword is valid only on the machine where the central manager is running. Using this keyword in a local configuration file has no effect.
2. It is recommended that you do not use **UserPrio** in the **SYSPRIO** expression, since user jobs are already ordered by **UserPrio**.
3. The string SYSPRIO can be used as both the name of an expression (SYSPRIO: *value*) and the name of a variable (SYSPRIO = *value*).

   To specify the expression to be used to calculate job priority you must use the syntax for the SYSPRIO expression. If the variable is mistakenly used for the SYSPRIO expression, which requires a colon (:) after the name, the job priority value will always be 0 because the SYSPRIO expression has not been defined.
4. When the **UserRunningJobs**, **GroupRunningJobs**, **UserQueuedJobs**, **GroupQueuedJobs**, **UserTotalJobs**, **GroupTotalJobs**, **GroupTotalShares**, **GroupUsedShares**, **UserTotalShares**, **UserUsedShares**, **GroupUsedBgShares**, **JobIsBlueGene**, and **UserUsedBgShares** variables are used to prioritize the queue based on current usage, you should also set **NEGOTIATOR_RECALCULATE_SYSPRIO_INTERVAL** so that the priorities are adjusted according to current usage rather than usage only at submission time.

Examples:

- Example 1

  This example creates a FIFO job queue based on submission time:

  ```
  SYSPRIO : 0 - (QDate)
  ```

- Example 2

  This example accounts for Class, User, and Group system priorities:

  ```
  SYSPRIO : (ClassSysprio * 100) + (UserSysprio * 10) + (GroupSysprio * 1) - (QDate)
  ```

- Example 3

  This example orders the queue based on the number of jobs a user is currently running. The user who has the fewest jobs running is first in the queue. You should set **NEGOTIATOR_RECALCULATE_SYSPRIO_INTERVAL** in conjunction with this **SYSPRIO** expression.

  ```
  SYSPRIO : 0 - UserRunningJobs
  ```

- Example 4

This example shows one possible way to set up the SYSPRIO expression for fair share scheduling. For those jobs whose owner has no unused shares ($(UserHasShares)= 0), that job priority depends only on QDate, making it a simple FIFO queue as in Example 1.

For those jobs whose owner has unused shares ($(UserHasShares)= 1), job priority depends not only on QDate, but also on a uniform boost of 31 536 000 (the equivalent to the job being submitted one year earlier). These jobs still have priority differences because of submit time differences. It is like forming two priority tiers: the higher priority tier for jobs with unused shares and the lower priority tier for jobs without unused shares.

```
SYSPRIO: 31536000 * $(UserHasShares) - QDate
```

- Example 5

  This example divides the jobs into three priority tiers:

  – Those jobs whose owner and group both have unused shares are at the top tier
  – Those jobs whose owner or group has unused shares are at the middle tier
  – Those jobs whose owner and group both have no shares remaining are at the bottom tier

  A user can submit two jobs to two different groups, the first job to a group with shares remaining and the second job to a group without any unused shares. If the user has unused shares, the first job will belong to the top tier and the second job will belong to the middle tier. If the user has no shares remaining, the first job will belong to the middle tier and the second job will belong to the bottom tier. The jobs in the top tier will be considered to run first, then the jobs in the middle tier, and lastly the jobs in the bottom tier.

  ```
  SYSPRIO: 31536000 * ($(UserHasShares)+$(GroupHasShares)) - (QDate)
  ```

  For more information related to using this keyword, see "Setting negotiator characteristics and policies" on page 47.

- Example 6

  This example show one possible way to prevent a job in user hold from increasing in priority relative to other jobs in the queue:

  ```
  SYSPRIO : 0 - QDate - UserHoldTime
  ```

SYSPRIO_THRESHOLD_TO_IGNORE_STEP

Specifies a threshold value for system priority. When the system priority assigned to a job step is less than the value set for this keyword, the scheduler ignores the job, which will remain in Idle state.

Syntax:

```
SYSPRIO_THRESHOLD_TO_IGNORE_STEP = integer
```

Any integer is a valid value.

Default value: INT_MIN

For more information related to using this keyword, see "Controlling the central manager scheduling cycle" on page 76.

TRACE

Specifies the debug control flags that are used to control the trace function of LoadLeveler.

Syntax:

```
TRACE = flags
```

The debug control flags that can be used are:

D_JOB_LIFECYCLE
Enables job lifecycle tracing if this flag was specified. If this debug flag is not used, job lifecycle information will not be logged (even if you request job tracing).

D_DISPATCHING_CYCLE
Turns on dispatching cycle tracing when LoadLeveler is started.

Default value: No default value is set.

TRUNC_KBDD_LOG_ON_OPEN
When **true**, specifies the log file is restarted with every invocation of the daemon.

Syntax:

```
TRUNC_KBDD_LOG_ON_OPEN = true | false
```

Default value: false

For more information related to using this keyword, see "Configuring recording activity and log files" on page 52.

TRUNC_MASTER_LOG_ON_OPEN
When **true**, specifies the log file is restarted with every invocation of the daemon.

Syntax:

```
TRUNC_MASTER_LOG_ON_OPEN = true | false
```

Default value: false

For more information related to using this keyword, see "Configuring recording activity and log files" on page 52.

TRUNC_NEGOTIATOR_LOG_ON_OPEN
When **true**, specifies the log file is restarted with every invocation of the daemon.

Syntax:

```
TRUNC_NEGOTIATOR_LOG_ON_OPEN = true | false
```

Default value: false

For more information related to using this keyword, see "Configuring recording activity and log files" on page 52.

TRUNC_REGION_MGR_LOG_ON_OPEN
When **true**, specifies the log file is restarted with every invocation of the daemon. This keyword is used by the resource manager component only.

Syntax:

```
TRUNC_REGION_MGR_LOG_ON_OPEN = true | false
```

Default value: false

For more information related to using this keyword, see "Configuring recording activity and log files" on page 52.

TRUNC_RESOURCE_MGR_LOG_ON_OPEN
When **true**, specifies the log file is restarted with every invocation of the daemon. This keyword is used by the resource manager component only.

Syntax:

```
TRUNC_RESOURCE_MGR_LOG_ON_OPEN = true | false
```

Default value: false

TRUNC_SCHEDD_LOG_ON_OPEN
When **true**, specifies the log file is restarted with every invocation of the daemon.

Syntax:

```
TRUNC_SCHEDD_LOG_ON_OPEN = true | false
```

Default value: false

For more information related to using this keyword, see "Configuring recording activity and log files" on page 52.

TRUNC_STARTD_LOG_ON_OPEN
When **true**, specifies the log file is restarted with every invocation of the daemon.

Syntax:

```
TRUNC_STARTD_LOG_ON_OPEN = true | false
```

Default value: false

For more information related to using this keyword, see "Configuring recording activity and log files" on page 52.

TRUNC_STARTER_LOG_ON_OPEN
When **true**, specifies the log file is restarted with every invocation of the daemon.

Syntax:

```
TRUNC_STARTER_LOG_ON_OPEN = true | false
```

Default value: false

For more information related to using this keyword, see "Configuring recording activity and log files" on page 52.

UPDATE_ON_POLL_INTERVAL_ONLY
Specifies whether or not the LoadLeveler **startd** daemons will send machine update transactions to the central manager. Normally the LoadLeveler startd daemons running on executing nodes will send transactions to the central manager to provide updates of machine information at various times. An update is sent every polling interval. The polling interval is calculated by multiplying the values for the two keywords, **POLLING_FREQUENCY** and **POLLS_PER_UPDATE**, specified in the LoadLeveler configuration file.

In addition, updates are sent at other times such as when new jobs are started and when jobs terminate on the executing node. If you have a large and highly active cluster (the workload consists of a large number of short running jobs), the normal method for updating the central manager can add excessive network traffic. **UPDATE_ON_POLL_INTERVAL_ONLY** can help reduce this source of network traffic.

When **true** is specified, the LoadLeveler startd daemon will only send machine updates to the central manager at every polling interval and not at other times.

Syntax:

```
UPDATE_ON_POLL_INTERVAL_ONLY = false | true
```

Default value: false

VACATE
Determines whether suspended jobs should be vacated.

Syntax:

```
VACATE: expression that evaluates to T or F (true or false)
```

When **T**, suspended LoadLeveler jobs are removed from the machine and placed back into the queue (provided you specify **restart=yes** in the job command file). If a checkpoint was taken, the job restarts from the checkpoint. Otherwise, the job restarts from the beginning.

Default value: No default value is set.

For information about time-related variables that you may use for this keyword, see "Variables to use for setting times" on page 291.

VM_IMAGE_ALGORITHM

Specifies the virtual memory algorithm, which is used for checking the image_size requirement. This keyword is used together with the **large_page** job command file keyword to specify which algorithm the central manager uses to decide whether a machine has enough virtual memory to run a job step.

This keyword is critical for job steps that must use Large Page memory (specified by the job command file keyword **large_page=M**). If the **VM_IMAGE_ALGORITHM** keyword is set to **FREE_PAGING_SPACE**, the Large Page job step will never be scheduled to run. This keyword must be set to **FREE_PAGING_SPACE_PLUS_FREE_REAL_MEMORY** to run Large Page jobs.

When **FREE_PAGING_SPACE** is specified, LoadLeveler considers only free paging space when determining if a machine has enough virtual memory to run a job step.

When **FREE_PAGING_SPACE_PLUS_FREE_REAL_MEMORY** is specified and the job step specifies:

- **large_page=N** (does not use Large Page memory), LoadLeveler considers free paging space and free regular memory when determining if a machine has enough virtual memory to run a job step.
- **large_page=Y** (uses Large Page memory, if available), LoadLeveler considers free paging space, free regular memory, and free Large Page memory when determining if a machine has enough virtual memory to run a job step, although Large Page memory is only considered for machines configured to exploit the Large Page feature.
- **large_page=M** (must use Large Page memory), LoadLeveler considers only Large Page memory when determining if a machine has enough virtual memory to run a job step. Only machines configured to exploit the Large Page feature are considered.

IBM suggests that you set this keyword to the value **FREE_PAGING_SPACE_PLUS_FREE_REAL_MEMORY** since more types of virtual memory are considered, increasing the chances of finding a machine with enough virtual memory to run the job step.

Syntax:

```
VM_IMAGE_ALGORITHM = FREE_PAGING_SPACE | FREE_PAGING_SPACE_PLUS_FREE_REAL_MEMORY
```

Default value: FREE_PAGING_SPACE

WALLCLOCK_ENFORCE

Specifies whether the job command file keyword **wall_clock_limit** will be enforced for this job. The **WALLCLOCK_ENFORCE** keyword is valid only when an external scheduler is enabled.

Syntax:

```
WALLCLOCK_ENFORCE = true | false
```

Default value: true

X_RUNS_HERE

Specifies whether the kbdd (keyboard) daemon runs on the host. If you want to run the kbdd daemon, specify **true**.

Syntax:

```
X_RUNS_HERE = true | false
```

Default value: false

User-defined keywords

This type of variable, which is generally created and defined by the user, can be named using any combination of letters and numbers.

A user-defined variable is set equal to values, where the *value* defines conditions, names files, or sets numeric values. For example, you can create a variable named **MY_MACHINE** and set it equal to the name of your machine named *iron* as follows:

```
MY_MACHINE = iron.ore.met.com
```

You can then identify the keyword using a dollar sign ($) and parenthesis. For example, the literal **$(MY_MACHINE)** following the definition in the previous example results in the automatic substitution of **iron.ore.met.com** in place of **$(MY_MACHINE)**.

User-defined definitions may contain references, enclosed in parenthesis, to previously defined keywords. Therefore:

```
A = xxx
C = $(A)
```

is a valid expression and the resulting value of **C** is *xxx*. Note that **C** is actually bound to **A**, not to its value, so that

```
A = xxx
C = $(A)
A = yyy
```

is also legal and the resulting value of **C** is *yyy*.

User-defined keywords can be specified in certain LoadLeveler configuration statements. Typically, the statements that can specify user-defined keywords are LoadLeveler expressions and statements that define file paths. The following list contains the LoadLeveler statements that can contain user-defined keywords:

acct_validation	**afs_getnewtoken**	**arch**
bin	**ckpt_execute_dir**	**comm**
continue	**custom_metric_command**	**execute**
ext_energy_policy_program	**global_history**	**history**
job_epilog	**job_prolog**	**job_user_epilog**
job_user_prolog	**kbdd**	**kbdd_coredump_dir**
kbdd_log	**kill**	**ll_rsh_command**

log	**machprio**	**master**
master_coredump_dir	**master_log**	**negotiator**
negotiator_coredump_dir	**negotiator_log**	**opsys**
process_tracking_extension	**region_mgr**	**region_mgr_coredump_dir**
region_mgr_log	**releasedir**	**reservation_history**
resource_mgr	**resource_mgr_coredump_dir**	**resource_mgr_log**
resource_mgr_spool	**schedd**	**schedd_coredump_dir**
schedd_log	**spool**	**start**
startd	**startd_coredump_dir**	**startd_log**
starter	**starter_coredump_dir**	**starter_log**
submit_filter	**suspend**	**sysprio**
vacate		

The sample configuration file shipped with the product defines and uses the following "user-defined" variables.

BackgroundLoad
Defines the variable **BackgroundLoad** and assigns to it a floating point constant. This might be used as a noise factor indicating no activity.

CPU_Busy
Defines the variable **CPU_Busy** and reassigns to it at each evaluation the Boolean value True or False, depending on whether the Berkeley one-minute load average is equal to or greater than the saturation level of 1.5.

CPU_Idle
Defines the variable **CPU_Idle** and reassigns to it at each evaluation the Boolean value True or False, depending on whether the Berkeley one-minute load average is equal or less than 0.7.

HighLoad
Is a keyword that the user can define to use as a saturation level at which no further jobs should be started.

HOUR
Defines the variable **HOUR** and assigns to it a constant integer value.

JobLoad
Defines the variable **JobLoad** which defines the load on the machine caused by running the job.

KeyboardBusy
Defines the variable **KeyboardBusy** and reassigns to it at each evaluation the Boolean value True or False, depending on whether the keyboard and mouse have been idle for fifteen minutes.

LowLoad
Defines the variable **LowLoad** and assigns to it the value of **BackgroundLoad**. This might be used as a restart level at which jobs can be started again and assumes only running 1 job on the machine.

mail
Specifies a local program you want to use in place of the LoadLeveler default mail notification method.

MINUTE
Defines the variable **MINUTE** and assigns to it a constant integer value.

StateTimer
Defines the variable **StateTimer** and reassigns to it at each evaluation the number of seconds since the current state was entered.

LoadLeveler variables

LoadLeveler provides variables that you can use in your configuration file statements. LoadLeveler variables are evaluated by the LoadLeveler daemons at various stages. They do not require you to use any special characters (such as a parenthesis or a dollar sign) to identify them.

Arch
Indicates the system architecture.

ClassSysprio
The priority for the class of the job step, defined in the class stanza in the administration file.

Default: 0

For additional information about using this variable, see the **SYSPRIO** keyword description.

Connectivity
The ratio of the number of active switch adapters on a node to the total number of switch adapters on the node. The value ranges from 0.0 (all switch adapters are down) to 1.0 (all switch adapters are active). A node with no switch adapters has a connectivity of 0.0. Connectivity can be used in a **MACHPRIO** expression to favor nodes that do not have any down switch adapters or in a job's **REQUIREMENTS** to require only nodes with a certain connectivity.

For additional information about using this variable, see the **MACHPRIO** keyword description.

ConsumableCpus
The number of **ConsumableCpus** currently available on the machine, if **ConsumableCpus** is defined in the configuration file keyword, **SCHEDULE_BY_RESOURCES**. If it is not defined in **SCHEDULE_BY_RESOURCES**, then it is equivalent to **Cpus**.

For additional information about using this variable, see the **MACHPRIO** keyword description.

ConsumableLargePageMemory
The amount of **ConsumableLargePageMemory**, in megabytes, currently available on the machine, if **ConsumableVirtualMemory** is defined in the **SCHEDULE_BY_RESOURCES** configuration file keyword. If it is not defined in **SCHEDULE_BY_RESOURCES**, then it is equal to the total amount of large page memory on the machine.

For additional information about using this variable, see the **MACHPRIO** keyword description.

ConsumableMemory
The amount of **ConsumableMemory**, in megabytes, currently available on the machine, if **ConsumableMemory** is defined in the configuration file keyword, **SCHEDULE_BY_RESOURCES**. If it is not defined in **SCHEDULE_BY_RESOURCES**, then it is equivalent to **Memory**.

For additional information about using this variable, see the **MACHPRIO** keyword description.

ConsumableVirtualMemory
The amount of **ConsumableVirtualMemory**, in megabytes, currently available on the machine, if **ConsumableVirtualMemory** is defined in the configuration file keyword, **SCHEDULE_BY_RESOURCES**. If it is not defined in **SCHEDULE_BY_RESOURCES**, then it is equivalent to **VirtualMemory**.

For additional information about using this variable, see the **MACHPRIO** keyword description.

Cpus
The number of processors of the machine, reported by the startd daemon.

For additional information about using this variable, see the **MACHPRIO** keyword description.

CurrentTime
The **UNIX date**; the current system time, in seconds, since January 1, 1970, as returned by the time() function.

CustomMetric
Sets a relative priority number for one or more machines, based on the value of the **CUSTOM_METRIC** keyword.

For additional information about using this variable, see the **MACHPRIO** keyword description.

Disk
The free disk space in kilobytes on the file system where the executables for the LoadLeveler jobs assigned to this machine are stored. This refers to the file system that is defined by the execute keyword.

For additional information about using this variable, see the **MACHPRIO** keyword description.

domain or domainname
Dynamically indicates the official name of the domain of the current host machine where the program is running. Whenever a machine name can be specified or one is assumed, a domain name is assigned if none is present.

EnteredCurrentState
The value of **CurrentTime** when the current state (START, SUSPEND, and so on) was entered.

FreeRealMemory
The amount of free real memory, in megabytes, on the machine. This value should track very closely with the "fre" value of the **vmstat** command and the "free" value of the **svmon -G** command (units are 4K blocks).

For additional information about using this variable, see the **MACHPRIO** keyword description.

GroupQueuedJobs
The number of job steps associated with a LoadLeveler group which are either running or queued. (That is, job steps which are in one of these states: Checkpointing, Preempted, Preempt Pending, Resume Pending, Running, Starting, Pending, or Idle.)

For additional information about using this variable, see the **SYSPRIO** keyword description.

GroupRunningJobs
The number of job steps for the LoadLeveler group which are in one of these states: Checkpointing, Preempted, Preempt Pending, Resume Pending, Running, Starting, or Pending.

For additional information about using this variable, see the **SYSPRIO** keyword description.

GroupSysprio
The priority for the group of the job step, defined in the group stanza in the administration file.

Default: 0

For additional information about using this variable, see the **SYSPRIO** keyword description.

GroupTotalJobs
The total number of job steps associated with this LoadLeveler group. Total job steps are all job steps reported by the **llq** command.

For additional information about using this variable, see the **SYSPRIO** keyword description.

GroupTotalShares
The total number of shares allocated to a group as specified by the **fair_shares** keyword in the group stanza.

For additional information about using this variable, see the **SYSPRIO** keyword description.

GroupUsedBgShares
The number of Blue Gene shares already used by a group or jobs owned by the group.

For additional information about using this variable, see the **SYSPRIO** keyword description.

GroupUsedShares
The number of shares already used by a group or jobs of the LoadLeveler group.

For additional information about using this variable, see the **SYSPRIO** keyword description.

host or hostname
Dynamically indicates the standard host name as returned by gethostname() for the machine where the program is running. **host** and **hostname** are equivalent, and contain the name of the machine without the domain name appended to it. If administrators need to specify the domain name in the configuration file, they may use **domain** or **domainname** along with **host** or **hostname**. For example:

```
$(host).$(domain)
```

JobIsBlueGene
Indicates whether the job whose priority is being calculated using the **SYSPRIO** keyword is a Blue Gene job.

For additional information about using this variable, see the **SYSPRIO** keyword description.

KeyboardIdle
The number of seconds since the keyboard or mouse was last used. It also includes any telnet or interactive activity from any remote machine.

LoadAvg
The Berkely one-minute load average, a measure of the CPU load on the system. The load average is the average of the number of processes ready to run or waiting for disk I/O to complete. The load average does not map to CPU time.

For additional information about using this variable, see the **MACHPRIO** keyword description.

MasterMachPriority
A value that is equal to 1 for nodes which are master nodes (those with **master_node_exclusive = true**); this value is equal to 0 for nodes which are not master nodes. Assigning a high priority to master nodes may help job scheduling performance for parallel jobs which require master node features.

For additional information about using this variable, see the **MACHPRIO** keyword description.

Memory
The size of real memory, in megabytes, of the machine, reported by the startd daemon.

For additional information about using this variable, see the **MACHPRIO** keyword description.

OpSys
Indicates the operating system on the host where the program is running. This value is automatically determined and should not be defined in the configuration file.

PagesFreed
The number of pages freed per second by the page replacement algorithm of the virtual memory manager.

For additional information about using this variable, see the **MACHPRIO** keyword description.

PagesScanned
The number of pages scanned per second by the page replacement algorithm of the virtual memory manager.

For additional information about using this variable, see the **MACHPRIO** keyword description.

QDate
The difference in seconds between the UNIX date when the job step enters the queue and the UNIX date when the negotiator daemon starts up.

For additional information about using this variable, see the **SYSPRIO** keyword description.

Speed
The relative speed of the machine, defined in a machine stanza in the administration file.

Default: 1

For additional information about using this variable, see the **MACHPRIO** keyword description.

State
The state of the startd daemon.

tilde
The home directory for the LoadLeveler user ID.

UserHoldTime
The total time that a job is in User Hold state.

For additional information about using this variable, see the **SYSPRIO** keyword description.

UserPrio
The user-defined priority of the job step, specified in the job command file with the **user_priority** keyword. The priority ranges from 0 to 100, with higher numbers corresponding to greater priority.

Default: 50

For additional information about using this variable, see the **SYSPRIO** keyword description.

UserQueuedJobs
The number of job steps either running or queued for the user. (That is, job steps that are in one of these states: Checkpointing, Preempted, Preempt Pending, Resume Pending, Running, Starting, Pending, or Idle.)

For additional information about using this variable, see the **SYSPRIO** keyword description.

UserRunningJobs
The number of job step steps for the user which are in one of these states: Checkpointing, Preempted, Preempt Pending, Resume Pending, Running, Starting, or Pending.

For additional information about using this variable, see the **SYSPRIO** keyword description.

UserSysprio
The priority of the user who submitted the job step, defined in the user stanza in the administration file.

Default: 0

For additional information about using this variable, see the **SYSPRIO** keyword description.

UserTotalJobs
The total number of job steps associated with this user. Total job steps are all job steps reported by the **llq** command.

For additional information about using this variable, see the **SYSPRIO** keyword description.

UserTotalShares
The total number of shares allocated to a user as specified by the **fair_shares** keyword in the user stanza.

For additional information about using this variable, see the **SYSPRIO** keyword description.

UserUsedBgShares
The number of Blue Gene shares already used by a user or jobs owned by the user.

For additional information about using this variable, see the **SYSPRIO** keyword description.

UserUsedShares
The number of shares already used by a user or jobs owned by the user.

For additional information about using this variable, see the **SYSPRIO** keyword description.

VirtualMemory
The size of available swap space (free paging space) on the machine, in kilobytes, reported by the startd daemon.

For additional information about using this variable, see the **MACHPRIO** keyword description.

Variables to use for setting dates

You can use the following date variables:

tm_mday
The number of the day of the month (1-31).

tm_mon
Number of months since January (0-11).

tm_wday
Number of days since Sunday (0-6).

tm_yday
Number of days since January 1 (0-365).

tm_year
The number of years since 1900 (0-9999). For example:

```
tm_year == 100
```

Denotes the year 2000.

tm4_year
The integer representation of the current year. For example:

```
tm4_year == 2010
```

Denotes the year 2010.

Variables to use for setting times

You can use the following time variables in the START, SUSPEND, CONTINUE, VACATE, and KILL expressions.

If you use these variables in the START expression and you are operating across multiple time zones, unexpected results may occur. This is because the negotiator daemon evaluates the START expressions and this evaluation is done in the time zone in which the negotiator resides. Your executing machine also evaluates the START expression and if your executing machine is in a different time zone, the results you may receive may be inconsistent. To prevent this inconsistency from occurring, ensure that both your negotiator daemon and your executing machine are in the same time zone.

tm_hour
The number of hours since midnight (0-23).

tm_isdst
Daylight Savings Time flag: positive when in effect, zero when not in effect,

negative when information is unavailable. For example, to start jobs between 5 PM and 8 AM during the month of October, factoring in an adjustment for Daylight Savings Time, you can issue:

```
START: (tm_mon == 9) && (tm_hour < 8) && (tm_hour > 17) && (tm_isdst = 1)
```

tm_min
Number of minutes after the hour (0-59).

tm_sec
Number of seconds after the minute (0-59).

Chapter 11. Administration keyword reference

For a file-based configuration, the administration file lists and defines the machines in the LoadLeveler cluster, as well as the characteristics of classes, users, groups, and clusters.

LoadLeveler does not prevent you from having multiple copies of administration files, but having only one administration file prevents confusion and avoids potential problems that might arise from having multiple files to update. To use only one administration file that is available to all machines in a cluster, you must place the file in a shared file system.

For the database configuration option, the definitions for the machines in the LoadLeveler cluster, as well as the characteristics of classes, users, groups, and clusters are kept in database tables.

Table 51 lists the administration file subtasks:

Table 51. Administration file subtasks

Subtask	Associated information (see . . .)
To find out what administrator tasks you can accomplish by using the administration file	Chapter 4, "Configuring the LoadLeveler environment," on page 39
To learn how to correctly specify the contents of an administration file	• "Administration file structure and syntax" • "Administration keyword descriptions" on page 298

Administration file structure and syntax

The administration file is called **LoadL_admin** and it lists and defines the *machine*, *machine_group*, *user*, *class*, *group*, *cluster* and *region*. For the database configuration option, there are one or more tables that correspond to each of these stanzas.

Machine and machine_group stanzas
Defines the roles that the machines in the LoadLeveler cluster play. See "Defining machines" on page 89 for more information.

User stanza
Defines LoadLeveler users and their characteristics. See "Defining users" on page 102 for more information.

Class stanza
Defines the characteristics of the job classes. To define characteristics that apply to specific users, user substanzas can be added within a class stanza. See "Defining classes" on page 94 and "Defining user substanzas in class stanzas" on page 99 for more information.

Group stanza
Defines the characteristics of a collection of users that form a LoadLeveler group. See "Defining groups" on page 103 for more information.

Cluster stanza
Defines the characteristics of a LoadLeveler cluster for use in a multicluster environment. See "Defining clusters" on page 104 for more information.

Region stanza
Defines the characteristics of a region in a LoadLeveler cluster. See "Defining regions" on page 106 for more information.

Stanzas have the following general format:

```
label: type = type_of_stanza
keyword1 = value1
keyword2 = value2
...
```

Substanzas have the following general format:

```
label: {
   type = type_of_stanza
   keyword1 = value1
   keyword2 = value2
   ...
   substanza_label: {
      type = type_of_substanza
      keyword3 = value3
   }
}
```

Keywords are *not* case sensitive. This means you can enter them in lower case, upper case, or mixed case.

The following is a simple example of an administration file illustrating several stanzas:

```
machine_a: type = machine
      central_manager = true       # defines this machine as the central manager

class_a: type = class
      priority = 50    # priority of this class

user_a: type  = user
      priority  = 50   # priority of this user

group_a: type = group
      priority  = 50   # priority of this group

adapter_a: type = adapter
      adapter_name = en0  #defines an adapter
```

The following is a simple example of an administration file illustrating a class stanza that contains user substanzas:

```
default:
     type = machine
     central_manager = false
     schedd_host = true

default:
     type = class
     wall_clocK-limit = 60:00, 30:00

parallel: {
     type = class

     # Allow at most 50 running jobs for class parallel
     maxjobs = 50

     # Allow at most 10 running jobs for any single
     # user of class parallel
     default: {
```

```
          type = user
          maxjobs = 10
     }

     # Allow user dept_head to run as many as 20 jobs
     # of class parallel
     dept_head: {
          type = user
          maxjobs = 20
     }
}

dept_head:
     type = user
     maxjobs = 30
```

Stanza characteristics

There are a number of characteristics associated with stanzas. The characteristics of a stanza are:

- Every stanza has a label associated with it. The label specifies the name you give to the stanza.
- Every stanza has a **type** field that specifies it as a machine, machine_group, user, class, group, cluster, or region stanza.
- New line characters are ignored. This means that separate parts of a stanza can be included on the same line. However, it is not recommended to have parts of a stanza cross line boundaries.
- White space is ignored, other than to delimit keyword identifiers. This eliminates confusion between tabs and spaces at the beginning of lines.
- A crosshatch sign (#) identifies a comment and can appear anywhere on the line. All characters following this sign on that line are ignored.
- Multiple stanzas of the same label are allowed, but only the first label is used.
- Default stanzas specify the default values for any keywords which are not otherwise specified. Each stanza type can have an associated default stanza. A default stanza must appear in the administration file ahead of any specific stanza entries of the same type. For example, a default class stanza must appear ahead of any specific class stanzas you enter.
- Stanzas can be nested within other stanzas (these are known as *substanzas*). See “Defining user substanzas in class stanzas” on page 99 for more information.
- The use of opening and closing braces ({ and }) to mark the beginning and end of a stanza is optional for stanzas that *do not* contain substanzas. A stanza that contains substanzas *must* be specified using braces as delimiting characters. Only user substanzas within class stanzas are supported. No types of stanzas other than class support substanzas and no types of stanzas other than user can be provided as substanzas within a class.
- If a syntax error is encountered, the remainder of the stanza is ignored and processing resumes with the next stanza.

Syntax for limit keywords

The syntax for setting a limit is:

```
limit_type = hardlimit,softlimit
```

For example:

```
core_limit = 120kb,100kb
```

To specify only a hard limit, you can enter, for example:

```
core_limit = 120kb
```

To specify only a soft limit, you can enter, for example:

```
core_limit = ,100kb
```

In a keyword statement, you cannot have any blanks between the numerical value (100 in the previous example) and the units (kb). Also, you cannot have any blanks to the left or right of the comma when you define a limit in a job command file.

For limit keywords that refer to a data limit — such as **data_limit**, **core_limit**, **file_limit**, **stack_limit**, **rss_limit**, **as_limit**, and **memlock_limit** — the hard limit and the soft limit are expressed as:

integer[.fraction][units]

The allowable units for these limits are:

```
b bytes
w words
kb kilobytes (2**10 bytes)
kw kilowords (2**12 bytes)
mb megabytes (2**20 bytes)
mw megawords (2**22 bytes)
gb gigabytes (2**30 bytes)
gw gigawords (2**32 bytes)
tb terabytes (2**40 bytes)
tw terawords (2**42 bytes)
pb petabytes (2**50 bytes)
pw petawords (2**52 bytes)
eb exabytes  (2**60 bytes)
ew exawords  (2**62 bytes)
```

If no units are specified for data limits, then bytes are assumed.

For limit keywords that refer to a number limit — such as **nproc_limit**, **locks_limit**, and **nofile_limit** — the hard limit and the soft limit are expressed as:

integer[.fraction][units]

The allowable units for these limits are:

```
K    Kilo     (2**10)
M    Mega     (2**20)
G    Giga     (2**30)
T    Tera     (2**40)
P    Peta     (2**50)
E    Exa      (2**60)
```

For limit keywords that refer to a time limit — such as **ckpt_time_limit**, **cpu_limit**, **job_cpu_limit**, and **wall_clock_limit** — the hard limit and the soft limit are expressed as:

[[hours:]minutes:]seconds[.fraction]

Fractions are rounded to seconds.

You can use the following character strings with all limit keywords except the **copy** keyword for **wall_clock_limit**, **job_cpu_limit**, and **ckpt_time_limit**:

rlim_infinity
: Represents the largest positive number.

unlimited
: Has same effect as **rlim_infinity**.

copy Uses the limit currently active when the job is submitted.

64-bit support for administration file keywords

Administrators can assign 64-bit integer values to selected keywords in the administration file. System resource limits, with the exception of CPU limits, are treated by LoadLeveler daemons and commands as 64-bit limits.

Table 52 describes 64-bit support for specific administration file keywords.

Table 52. Notes on 64-bit support for administration file keywords

Keyword	Stanza	Notes
as_limit	Class	See the notes for **core_limit** and **data_limit**.
core_limit **data_limit**	Class	64-bit integer values can be assigned to these limits. Fractional specifications are allowed and will be converted to 64-bit integer values. Unit specifications are accepted and can be one of the following: b, w, kb, kw, mb, mw, gb, gw, tb, tw, pb, pw, eb, ew. **Example:** `core_limit = 8gb,4.25gb`
default_resources **default_node_resources**	Class	Consumable resources associated with the **default_resources** and **default_node_resources** keywords can be assigned 64-bit integer values. Fractional specifications are not allowed. Unit specifications are valid only when specifying the values of the predefined **ConsumableMemory**, **ConsumableVirtualMemory**, and **ConsumableLargePageMemory** resources. **Example:** `default_resources = ConsumableVirtualMemory(12 gb)db2_license(112)`
file_limit	Class	See the notes for **core_limit** and **data_limit**.
locks_limit	Class	See the notes for **nproc_limit**.
memlock_limit	Class	See the notes for **core_limit** and **data_limit**.
nofile_limit	Class	See the notes for **nproc_limit**.
nproc_limit	Class	64-bit integer values can be assigned to these limits. Fractional specifications are allowed and will be converted to 64-bit integer values. Unit specifications is number. Unit specification is a number that can be used in conjunction with the following abbreviations: **K** kilo **M** mega **G** giga **T** tera **P** peta **E** exa **Examples:** `nproc_limit = 1000,285`

resources	Machine	Consumable resources associated with the **resources** keyword can be assigned 64-bit integer values. Fractional specifications are not allowed. Unit specifications are valid only when specifying the values of the predefined **ConsumableMemory**, **ConsumableVirtualMemory**, and **ConsumableLargePageMemory** resources.
		
		Examples:
		`resources = spice2g6(9123456789012) ConsumableMemory(10 gw)`
		`resources = ConsumableVirtualMemory(15 pb) db2_license(1234567890)`

Table 52. Notes on 64-bit support for administration file keywords (continued)

Keyword	Stanza	Notes
rss_limit	Class	See the notes for **core_limit** and **data_limit**.
stack_limit		**Example:** `rss_limit = 1.25eb,3.33pw`

64-bit limits on Linux systems

Applications managed by LoadLeveler for AIX can be 64-bit applications if the hardware architecture on which AIX is running is capable of supporting 64-bit processes.

Resource limits, such as data limits and stack limits, can be 64-bit limits. When a value of *unlimited* is specified for a process limit (**cpu_limit** excepted) in the LoadLeveler administration file or job command file, the AIX version of LoadLeveler stores this value internally as `INT64_MAX`. Before starting the user job, **LoadL_starter** sets the appropriate limit to this value. This behavior is correct because, on AIX, `RLIM64_INFINITY` is the same as `INT64_MAX (= 0x7FFFFFFFFFFFFFFFLL)`.

On Linux systems, `RLIM64_INFINITY` is equal to `UINT64_MAX (= 0xFFFFFFFFFFFFFFFFULL)`. To maintain compatibility with AIX, LoadLeveler for Linux also stores *unlimited* internally as `INT64_MAX`. However, **LoadL_starter** on Linux sets all process limits (**cpu_limit** excepted) that are in the range (`INT64_MAX`, `UINT64_MAX`) to `UINT64_MAX` before starting the jobs managed by LoadLeveler.

For historical reasons, LoadLeveler for AIX treats the hard and soft time limits, such as **cpu_limit**, **job_cpu_limit**, and **wall_clock_limit**, as 32-bit limits and *unlimited* means `INT32_MAX`. For consistency reasons, LoadLeveler for Linux assumes the same behavior.

Administration keyword descriptions

This topic contains an alphabetical list of the LoadLeveler administration keywords.

account

Specifies a list of account numbers available to a user submitting jobs.

Syntax:

`account =`*`list`*

Where *list* is a blank-delimited list of account numbers that identifies the account numbers a user can use when submitting jobs.

Default: A null list.

admin

Specifies a list of administrators for a group or class.

Syntax:

`admin =` *`list`*

Where *list* is a blank-delimited list of administrators for either this class or this group, depending on whether this keyword appears in a class or group stanza, respectively. These administrators can hold, release, and cancel jobs in this class or this group.

adapter_list
Specifies a list of adapters to be used on the machine.

Syntax:

```
adapter_list = list1 list2 ...
```

or

```
adapter_list = none
```

Where the *list* of adapters consists of interface names in the order that will be used for scheduling. If the **adapter_list** keyword is not specified, all adapters that were found dynamically will be used for scheduling. If **adapter_list** is set to **none**, no adapters will be used by this machine. The **adapter_list** cannot be set to a blank value.

as_limit
Specifies the hard limit, soft limit, or both for the address space to be used by each process of the submitted job.

Syntax:

```
as_limit = hardlimit,softlimit
```

Examples:

```
as_limit = 125621          # hardlimit = 125621 bytes
as_limit = 5621kb          # hardlimit = 5621 kilobytes
as_limit = 2mb             # hardlimit = 2 megabytes
as_limit = 2.5mw           # hardlimit = 2.5 megawords
as_limit = unlimited       # hardlimit = 9,223,372,036,854,775,807 \
                             bytes (X'7FFFFFFFFFFFFFFF')
as_limit = rlim_infinity   # hardlimit = 9,223,372,036,854,775,807 \
                             bytes (X'7FFFFFFFFFFFFFFF')
```

For additional information about limit keywords, see the following topics:
- "Syntax for limit keywords" on page 295
- "Using limit keywords" on page 94

central_manager
Determines whether the machine is the LoadLeveler central manager.

Syntax:

```
central_manager = true| false | alt
```

Where:

- **true** designates this machine as the LoadLeveler central manager host, where the negotiator daemon runs. You must specify one and only one machine stanza identifying the central manager. For example:

  ```
  machine_a: type = machine
  central_manager = true
  ```

- **false** specifies that this machine is not the central manager.
- **alt** specifies that this machine can serve as an alternate central manager in the event that the primary central manager is not functioning. For more information on recovering if the primary central manager is not operating, refer to "What happens if the central manager isn't operating?" on page 399. Submit-only machines cannot have their machine stanzas set to this value.

  If you are going to select machines to serve as alternate central managers, you should look at the following keywords in the configuration file:

  – **FAILOVER_HEARTBEAT_INTERVAL**

  – **FAILOVER_HEARTBEAT_RETRIES**

For information on setting these keywords, see "Specifying alternate central managers" on page 48.

Default: false

Note: This keyword is deprecated, use the **central_manager_list** keyword instead.

ckpt_dir

Specifies the directory to be used for checkpoint files for jobs that did not specify this directory in the job command file.

The actual directory used to store the checkpoint files is a combination of the value of this keyword and the value of the **ckpt_subdir** keyword.

Syntax:

```
ckpt_dir = directory
```

Where *directory* is the directory location to be used for checkpoint files that did not have a directory name specified in the job command file. If the value specified does not have a fully qualified directory path (including the beginning forward slash), the initial working directory will be inserted before the specified value.

The value specified by the **ckpt_dir** keyword is only used when the **ckpt_subdir** keyword does not contain a full path name in the job command file or is not specified in the job command file.

Default: Initial working directory

ckpt_time_limit

Specifies the hard limit, soft limit, or both limits for the elapsed time that checkpointing a job can take.

Syntax:

```
ckpt_time_limit = hardlimit,softlimit
```

Where *hardlimit,softlimit* defines the maximum time that checkpointing a job can take. When LoadLeveler detects that the softlimit has been exceeded, it attempts to end the checkpoint and allow the job to continue. If this is not possible, and the hard limit is exceeded, LoadLeveler will terminate the job. The start time of the checkpoint is defined as the time when the Startd daemon receives status from the starter that a checkpoint has started.

Default: Unlimited

Examples:

```
ckpt_time_limit = 30:45          #hardlimit - 30 minutes 45 seconds
ckpt_time_limit = 30:45,25:00    #hardlimit - 30 minutes 45 seconds
                                 #softlimit - 25 minutes
```

For additional information about limit keywords, see the following topics:

- "Syntax for limit keywords" on page 295
- "Using limit keywords" on page 94

class

Determines whether a machine will accept jobs of a certain job class. For parallel jobs, you must define a class instance for each task you want to run on a node using the format, **class** = *class_name (count)*. This format defines the **class** names using a statement that names the classes and sets the number of tasks for each class in parenthesis.

With this format, the following rules apply:
- Each class can have only one entry
- If a class has more than one entry or there is a syntax error, the entire **class** statement will be ignored
- If the **class** statement has a blank value or is not specified, it will be defaulted to **No_Class (1)**
- The number of instances for a class specified inside the parenthesis **( )** must be an unsigned integer. If the number specified is 0, it is correct syntactically, but the class will not be defined in LoadLeveler
- If the number of instances for all classes in the **class** statement are 0, the default **No_Class(1)** will be used

Note: The class names list is blank delimited.

For a LoadLeveler job to run on a machine, the machine must have a vacancy for the class of that job. If the machine is configured for only one **No_Class** job and a LoadLeveler job is already running there, then no further LoadLeveler jobs are started on that machine until the current job completes.

You can have a maximum of 1024 characters in the class statement. You cannot use **allclasses** or **data_stage** as a class name, since these are reserved LoadLeveler keywords.

You can assign multiple classes to the same machine by specifying the classes in the LoadLeveler administration file (called **LoadL_admin**). The classes, themselves, should also be defined in the administration file. See "Setting up a single machine to have multiple job classes" on page 415 and "Defining classes" on page 94 for more information on classes.

Syntax:

```
class = class_name (count) ...
```

Default value: No_Class(1)

class_comment

Text characterizing the class.

Syntax:

```
class_comment = "string"
```

Where *string* is text characterizing the class. The comment string associated with this keyword cannot contain an equal sign (**=**) or a colon (**:**) character. The length of the string cannot exceed 1024 characters.

Default: No default value is set.

collective_groups

Requests the Collective Acceleration Unit (CAU) groups for the specified protocol instances of the job.

Syntax:

```
collective_groups = number
```

The value of the collective groups must be greater than or equal to zero. The value specified for the **collective_groups** keyword in the job command file overwrites any value specified for the **collective_groups** keyword in the administration file. If the job is not sharing the nodes with other jobs, then the protocol instance of the job will be allocated at least *number* CAU groups. Additional CAUs can be allocated to the job step if additional CAU groups are available on the node and the node is not shared with other jobs. If the job is

sharing the node with other jobs, then exactly *number* CAU groups are allocated to each protocol instance of the job.

Default value: The default value varies depending on whether the job shares the nodes with other jobs or not. If the job is not sharing the nodes with other jobs, then all the CAU groups are allocated to all the protocol instances of the job proportionally. If the job is sharing the nodes with other jobs, then zero CAU groups are allocated to the protocol instances of the job.

core_limit

Specifies the hard limit, soft limit, or both limits for the size of a core file a job can create.

Syntax:

```
core_limit = hardlimit,softlimit
```

Examples:

```
core_limit = unlimited
core_limit = 30mb
```

For additional information about limit keywords, see the following topics:
- "Syntax for limit keywords" on page 295
- "Using limit keywords" on page 94

cpu_limit

Specifies hard limit, soft limit, or both limits for the CPU time to be used by each individual process of a job step.

Syntax:

```
cpu_limit = hardlimit,softlimit
```

For example, if you impose a **cpu_limit** of five hours and you have a job step composed of five processes, each process can consume five CPU hours; the entire job step can therefore consume 25 total hours of CPU.

Examples:

```
cpu_limit = 12:56:21       # hardlimit = 12 hours 56 minutes 21 seconds
cpu_limit = 56:00,50:00    # hardlimit = 56 minutes 0 seconds
                           # softlimit = 50 minutes 0 seconds
cpu_limit = 1:03           # hardlimit = 1 minute 3 seconds
cpu_limit = unlimited      # hardlimit = 2,147,483,647 seconds
                           # (X'7FFFFFFF')
cpu_limit = rlim_infinity  # hardlimit = 2,147,483,647 seconds
                           # (X'7FFFFFFF')
cpu_limit = copy           # current CPU hardlimit value on the
                           # submitting machine.
```

For additional information about limit keywords, see the following topics:
- "Syntax for limit keywords" on page 295
- "Using limit keywords" on page 94

cpu_speed_scale

Determines whether CPU time is normalized according to machine speed.

Syntax:

```
cpu_speed_scale = true | false
```

Where **true** specifies that CPU time (which is used, for example, in setting limits, in accounting information, and reported by the **llq -x** command), is in normalized units for each machine. **false** specifies that CPU time is in native

units for each machine. For an example of using this keyword to normalize accounting information, see "Example: Setting up job accounting files" on page 71.

Default: false

data_limit

Specifies hard limit, soft limit, or both for the data segment to be used by each process of the submitted job.

Syntax:

```
data_limit = hardlimit,softlimit
```

Examples:

```
data_limit = 125621         # hardlimit = 125621 bytes
data_limit = 5621kb         # hardlimit = 5621 kilobytes
data_limit = 2mb            # hardlimit = 2 megabytes
data_limit = 2.5mw          # hardlimit = 2.5 megawords
data_limit = unlimited      # hardlimit = 9,223,372,036,854,775,807 bytes
                            # (X'7FFFFFFFFFFFFFFF')
data_limit = rlim_infinity  # hardlimit = 9,223,372,036,854,775,807 bytes
                            # (X'7FFFFFFFFFFFFFFF')
data_limit = copy           # copy data hardlimit value from
                            # submitting machine.
```

For additional information about limit keywords, see the following topics:

- "Syntax for limit keywords" on page 295
- "Using limit keywords" on page 94

default_class

Specifies a class name that is the default value assigned to jobs submitted by users for which no class statement appears.

Syntax:

```
default_class = list
```

Where *list* is a blank-delimited list of class names used for jobs which do not include a **class** statement in the job command file. If you specify only one default class name, this class is assigned to the job. If you specify a list of default class names, LoadLeveler searches the list to find a class which satisfies the resource limit requirements. If no class satisfies these requirements, LoadLeveler rejects the job.

Suppose a job requests a CPU limit of 10 minutes. Also, suppose the default class list is `default_class = short long`, where `short` is a class for jobs up to five minutes in length and `long` is a class for jobs up to one hour in length. LoadLeveler will select the `long` class for this job because the `short` class does not have sufficient resources.

Default: If no **default_class** is specified in the user stanza, or if there is no user stanza at all, then jobs submitted without a **class** statement are assigned to the **default_class** that appears in the default user stanza. If you do not define a **default_class**, jobs are assigned to the class called **No_Class**.

default_group

Specifies the default group name to which the user belongs.

Syntax:

```
default_group = group_name
```

Where *group_name* is the default group assigned to jobs submitted by the user.

If you specify **default_group = Unix_Group**, LoadLeveler sets the user's LoadLeveler group to the user's current UNIX group.

Default: If a **default_group** statement does not appear in the user stanza, or if there is no user stanza at all, then jobs submitted by the user without a **group** statement are assigned to the **default_group** that appears in the default user stanza. If you do not define a **default_group**, jobs are assigned to the group called **No_Group**.

default_interactive_class

Specifies a class to which interactive jobs are assigned for jobs submitted by users who do not specify a class using the LOADL_INTERACTIVE_CLASS variable. You can specify only one default interactive class name.

Syntax:

```
default_interactive_class = class_name
```

Where *class_name* is the class to which an interactive job submitted by this user is assigned if the user does not specify a class using the LOADL_INTERACTIVE_CLASS environment variable.

Default: If you do not set a **default_interactive_class** value in the user stanza, or if there is no user stanza at all, then interactive jobs submitted without a **class** statement are assigned to the **default_interactive_class** that appears in the default user stanza. If you do not define a **default_interactive_class**, interactive jobs are assigned to the class called **No_Class**.

See "Examples: User stanzas" on page 102 for more information on how LoadLeveler assigns a default interactive class to jobs.

default_network

Specifies the default communication protocols, adapters, and their characteristics provided there is no network statement in the job step.

Note: Default values for **collective_groups**, **imm_send_buffers**, and **endpoints** cannot be specified in the **default_network** statement. **MPI**, **LAPI**, and **MPI_LAPI** are the only protocols that can be specified. The **default_network** statement is not supported when using the database configuration option.

Syntax:

```
default_network.protocol = type[, usage[, mode[,comm_level[, instances=<number
|max> [, rcxtblocks=number]]]]]
```

where:

protocol

Specifies the communication protocols that are used with an adapter, and can be the following:

MPI Specifies the message passing interface (MPI). You can specify in a class stanza both **default_network.MPI** and **default_network.LAPI**.

LAPI Specifies the low-level application programming interface (LAPI). You can specify in a class stanza both **default_network.MPI** and **default_network.LAPI**.

MPI_LAPI

Specifies sharing adapter windows between MPI and LAPI. When you specify **default_network.MPI_LAPI** in a class stanza, you cannot specify any other network statements in that class stanza.

type This field is required and specifies one of the following:

sn_single
When used for switch adapters, it specifies that LoadLeveler use a common, single switch network.

sn_all Specifies that striped communication should be used over all available switch networks. The networks specified must be accessible by all machines selected to run the job. For more information on striping, see "Submitting jobs that use striping" on page 188.

The following are optional and if omitted their position must be specified with a comma:

usage Specifies whether the adapter can be shared with tasks of other job steps. Possible values are **shared**, which is the default, or **not_shared**. If **not_shared** is specified, LoadLeveler can only guarantee that the adapter will not be shared by other jobs running on the same OSI. If the adapter is shared by more than one OSI, LoadLeveler cannot guarantee that the adapter is not shared with jobs running on a different OSI.

mode Specifies the communication subsystem mode used by the communication protocol that you specify, and can be either **IP** (Internet Protocol), which is the default, or **US** (User Space). Note that each instance of the US mode requested by a task running on switch adapters requires an adapter window. For example, if a task requests both the MPI and LAPI protocols such that both protocol instances require US mode, two adapter windows will be used.

comm_level

Note: This keyword is obsolete and will be ignored, however it is being retained for compatibility and because the parameters in the **default_network** statement are positional.

The **comm_level** keyword should be used to suggest the amount of inter-task communication that users *expect* to occur in their parallel jobs. This suggestion is used to allocate adapter device resources. Specifying a level that is higher than what the job actually needs will not speed up communication, but may make it harder to schedule a job (because it requires more resources). The **comm_level** keyword can only be specified with **US** mode. The three communication levels are:

LOW Implies that minimal inter-task communication will occur.

AVERAGE
This is the default value. Unless you know the specific communication characteristics of your job, the best way to determine the comm_level is through trial-and-error.

instances=<*number***|max>**
If **instances** is specified as a number, it indicates the number of parallel communication paths made available to the protocol on each network. The number actually used will depend on the implementation of the protocol subsystem. If **instances** is specified by **max**, the actual value used is determined by the **MAX_PROTOCOL_INSTANCES** in the class when the job is submitted. The default value for **instances** is 1.

For the best performance set **MAX_PROTOCOL_INSTANCES** so that the communication subsystem uses every available adapter before it reuses any of the adapters.

rcxtblocks=*number*
Integer value specifying the number of user rCxt blocks requested for each window used by the associated protocol.

Note: Use of this keyword will prevent adapters from the SP Switch2 family from being used by the job.

default_node_resources
Specifies quantities of the consumable resources consumed by each node of a job step provided that a **node_resources** keyword is not coded for the step in the job command file. If a **node_resources** keyword is coded for a job step, it overrides any default node resources associated with the associated job class. The resources must be machine resources; floating resources cannot be assigned with the **node_resources** keyword.

Syntax:

```
default_node_resources = name(count) name(count) ... name(count)
```

The administrator defines the name and count for **default_node_resources**. In addition, *name(count)* could be **ConsumableCpus**(*count*), **ConsumableMemory**(*count units*), **ConsumableLargePageMemory***(count units)*, or **ConsumableVirtualMemory**(*count units*).

The **ConsumableMemory**, **ConsumableVirtualMemory**, and **ConsumableLargePageMemory** can be specified with both a *count* and *units*. **ConsumableMemory** or **ConsumableVirtualMemory** specified resource *count* must be an integer greater than zero. **ConsumableLargePageMemory** specified resource *count* must be an integer greater than or equal to zero. The allowable *units* are those normally used with LoadLeveler data limits:

```
b bytes
w words
kb kilobytes (2**10 bytes)
kw kilowords (2**12 bytes)
mb megabytes (2**20 bytes)
mw megawords (2**22 bytes)
gb gigabytes (2**30 bytes)
gw gigawords (2**32 bytes)
tb terabytes (2**40 bytes)
tw terawords (2**42 bytes)
pb petabytes (2**50 bytes)
pw petawords (2**52 bytes)
eb exabytes  (2**60 bytes)
ew exawords  (2**62 bytes)
```

The **ConsumableMemory**, **ConsumableVirtualMemory**, and **ConsumableLargePageMemory** values are stored in MB (megabytes) and are rounded up. For **ConsumableMemory** or **ConsumableVirtualMemory** the smallest amount that you can request is 1 MB. If no units are specified, then megabytes are assumed. Resources defined here that are not in the **SCHEDULE_BY_RESOURCES** list in the global configuration file will not affect the scheduling of the job.

default_resources
Specifies the default amount of resources consumed by a task of a job step, provided that the **resources** or **dstg_resources** keyword is not coded for the step in the job command file. If a resources keyword is coded for a job step, then it overrides any **default resources** associated with the associated job class.

Syntax:

```
default_resources = name(count) name(count)...name(count)
```

The administrator defines the name and count values for **default_resources**. In addition, *name(count)* could be **ConsumableCpus***(count)*, **ConsumableMemory***(count units)*, **ConsumableVirtualMemory***(count units)*, or **ConsumableLargePageMemory***(count units)*.

The **ConsumableMemory**, **ConsumableVirtualMemory**, and **ConsumableLargePageMemory** can be specified with both a *count* and *units*. **ConsumableMemory** or **ConsumableVirtualMemory** specified resource *count* must be an integer greater than zero. **ConsumableLargePageMemory** specified resource *count* must be an integer greater than or equal to zero. The allowable *units* are those normally used with LoadLeveler data limits:

```
b bytes
w words
kb kilobytes (2**10 bytes)
kw kilowords (2**12 bytes)
mb megabytes (2**20 bytes)
mw megawords (2**22 bytes)
gb gigabytes (2**30 bytes)
gw gigawords (2**32 bytes)
tb terabytes (2**40 bytes)
tw terawords (2**42 bytes)
pb petabytes (2**50 bytes)
pw petawords (2**52 bytes)
eb exabytes  (2**60 bytes)
ew exawords  (2**62 bytes)
```

The **ConsumableMemory** and **ConsumableVirtualMemory** values are stored in MB (megabytes) and are rounded up. Therefore, the smallest amount of **ConsumableMemory** or **ConsumableVirtualMemory** that you can request is 1 MB. If no units are specified, then megabytes are assumed. Resources defined here that are not in the **SCHEDULE_BY_RESOURCES** list in the global configuration file will not effect the scheduling of the job.

default_wall_clock_limit

Sets a default value for jobs not specifying a wall clock limit in the job command file. The **wall_clock_limit** keyword serves only as the maximum value allowed for the class. The **default_wall_clock_limit** value can be overridden by a job using the **wall_clock_limit** job command file keyword, but that limit cannot exceed the **wall_clock_limit** configured in the class stanza.

Note: If **default_wall_clock_limit** is not specified, it will be assigned the value of **wall_clock_limit** for the same class.

Syntax:

default_wall_clock_limit = *hardlimit*,*softlimit*

An example is:

```
default_wall_clock_limit = 5:00,4:30
```

For additional information about limit keywords, see the following topics:
- "Syntax for limit keywords" on page 295
- "Using limit keywords" on page 94

dstg_max_starters

Specifies a machine-specific limit on the number of data staging initiators. Since each task of a data staging job step consumes one initiator from the **data_stage** class on the specified machine, **dstg_max_starters** provides the maximum number of data staging tasks that can run at the same time on the machine.

Syntax:

```
dstg_max_starters = number
```

Notes:

1. If you have not set the **dstg_max_starters** value in either the global or local configuration files, there will not be any data staging initiators on the specified machine. In this configuration, the executing machine will not be allowed to perform data staging tasks.
2. The value specified for **dstg_max_starters** will be the number of initiators available for the built-in **data_stage** class on that machine.
3. The value specified for **dstg_max_starters** will not limit the value specified for **dstg_max_starters**.

Default value: 0

endpoints

Specifies the number of endpoints that can be used by each task per protocol instance.

Syntax:

```
endpoints = 1 | 2 | 4 | 8 | 16 | 32 | 64 | 128
```

where:

number

Must be a power of 2 and no greater than 128 (that is, from {1, 2, 4, 8, 16, 32, 64, 128}). If the value specified is not a power of 2, the next higher power of 2 is used and a warning message is issued. The value of the **endpoints** keyword is inherited to all the protocol instances of the jobs.

Default value: One endpoints value is assigned for each task per protocol instance.

env_copy

Specifies a default value for the job command file **env_copy** keyword for the class, group or user stanza containing the keyword.

Syntax:

```
env_copy = all | master
```

Table 53 states the value that LoadLeveler uses depending on the combination of values set in the user, group, or class stanzas.

Table 53. Summary of possible values set for the **env_copy** *keyword in the administration file*

env_copy keyword setting in applicable stanzas in the administration file	**Resulting LoadLeveler default behavior for copying the job environment**
All stanzas that set the **env_copy** keyword specify **env_copy** = master	master becomes the default value for the job command file **env_copy** keyword.
One or more stanzas explicitly set **env_copy** = all	all becomes the default value for the job command file **env_copy** keyword.
The **env_copy** keyword is not specified in any stanza	

Default value: No default value is set.

For more information, see:

- The job command file env_copy keyword description.
- "Steps for reducing job launch overhead for parallel jobs" on page 111.

exclude_bg

Specifies that jobs using this class cannot run using the specified list of Blue Gene resources.

Syntax:

```
exclude_bg = list
```

Where *list* is a blank-delimited list of Blue Gene resources that cannot be used by jobs in this class.

An item on the *list* can be the name of a midplane (for example, R00-M0), the name of a rack (for example, R00), or the name of a row of racks (for example, R0). Other types of Blue Gene resources are not subject to any restrictions by the **include_bg** or **exclude_bg** keyword.

If both **exclude_bg** and **include_bg** are specified, **exclude_bg** takes precedence.

Default: The default is that no Blue Gene resources are excluded.

Examples:

In a Blue Gene/Q system, if job class **ClassB** has:

```
exclude_bg = R0 R11 R32-M0
```

the jobs in **ClassB** cannot use racks starting with "R0" (which includes midplanes R00, R01, R02, and so on), rack R11 (which includes midplanes R11–M0 and R11–M1), and midplane R32–M0.

exclude_classes

exclude_classes can be specified within a cluster stanza.

Specifies a blank-delimited list of one or more job classes that will not accept remote jobs within the cluster.

Syntax:

```
exclude_classes = class_name[(cluster_name)] ...
```

Where *class_name* specifies a class to be excluded and *cluster_name* can be used to specify that remote jobs from *cluster_name* submitted under *class_name* will be excluded but any other jobs submitted under *class_name* from other clusters will be allowed.

Do not specify a list of **exclude_classes** and **include_classes**. Only one of these keywords can be used within any cluster stanza. **exclude_classes** takes precedence over **include_classes** if both are specified.

Default: The default is that no classes are excluded.

exclude_groups

exclude_groups can be specified within a class stanza and a cluster stanza.

Class stanza:

When used within a class stanza, **exclude_groups** specifies a list of group names identifying those who cannot submit jobs of a particular class.

Syntax:

```
exclude_groups = list
```

Where *list* is a blank-delimited list of groups who are *not* allowed to submit jobs of *class name*.

This list can contain individual user names. To allow a list of users to be included with the list of group names, add a plus sign (+) to each user name that you add to the list. LoadLeveler treats these names as implicit groups.

For example, to add user **mike** to a list of group names, specify:

```
exclude_groups = prod +mike
```

If the string **+mike** is also the actual name of a group stanza, LoadLeveler treats this name as a group, not an implicit group. In this case, LoadLeveler will not prevent user **mike** from submitting jobs to this class unless the user is a member of the **prod** or **+mike** group.

If this keyword is specified, this list limits groups and users of that class to those on the list.

Do not specify both a list of included groups and a list of excluded groups. Only one of these may be used for any class stanza. **exclude_groups** takes precedence over **include_groups** if both are specified.

Default: The default is that no groups are excluded.

Cluster stanza:

When used within a cluster stanza, **exclude_groups** specifies a blank-delimited list of one or more groups that will not accept remote jobs within the cluster.

Syntax:

```
exclude_groups = group_name[(cluster_name)] ...
```

Where *group_name* specifies a group that is not allowed to submit remote jobs and *cluster_name* can be used to specify that remote jobs from *cluster_name* submitted under *group_name* will be excluded but any other jobs submitted under *group_name* from other clusters will be allowed.

Do not specify a list of **exclude_groups** and **include_groups**. Only one of these may be used within any cluster stanza. **exclude_groups** takes precedence over **include_groups** if both are specified.

Default: The default is that no groups are excluded.

exclude_users

exclude_users may be specified within a class, group, and cluster stanza.

Class or group stanza:

When used within a class or group stanza **exclude_users** specifies a list of user names identifying those who cannot submit jobs of a particular class or who are not members of the group.

Syntax:

```
exclude_users = list
```

The definition of this keyword varies slightly, depending on the type of administration file stanza in which the keyword appears:

- In a class stanza: *list* is a blank-delimited list of users who are *not* permitted to submit jobs of *class_name*.
- In a group stanza: *list* is a blank-delimited list of users who do not belong to the group.

Do not specify both a list of included users and a list of excluded users. Only one of these may be used for any class or group. **exclude_users** takes precedence over **include_users** if both are specified. In a class stanza, **exclude_users** also takes precedence over any user substanzas.

Default: The default is that no users are excluded.

Cluster stanza:

When used within a cluster stanza, **exclude_users** specifies a blank-delimited list of one or more users who cannot submit jobs to the cluster.

Syntax:

```
exclude_users = user_name[(cluster_name)] ...
```

Where *user_name* specifies a user that is not allowed to submit remote jobs and *cluster_name* can be used to specify that remote jobs from *cluster_name* submitted under the *user_name* will be excluded but any other jobs submitted under that *user_name* from other clusters will be allowed.

Do not specify a list of **exclude_users** and **include_users**. Only one of these may be used within any cluster stanza. **exclude_users** takes precedence over **include_users** if both are specified.

Default: The default is that no users are excluded.

fair_shares

Specifies the number of shares allocated to jobs of this user or group for fair share scheduling. If the user or group stanza does not specify **fair_shares**, or if there is no user or group stanza at all, the value in the default user or group stanza is used (which defaults to zero if not explicitly specified). The user or group has this number of shares of the cluster CPU resources as well as this number of shares of the Blue Gene resources (if Blue Gene resources are also available in the cluster).

Syntax:

```
fair_shares = number
```

For additional information about the fair share scheduling keyword, see "Using fair share scheduling" on page 158.

feature

Specifies an optional characteristic to use to match jobs with machines. You can specify unique characteristics for any machine using this keyword. When evaluating job submissions, LoadLeveler compares any required features specified in the job command file to those specified using this keyword. You can have a maximum of 1024 characters in the feature statement.

Syntax:

```
feature = string ...
```

Default value: No default value is set.

Example: If a machine has licenses for installed products ABC and XYZ in the local configuration file, you can enter the following:

```
feature = abc xyz
```

When submitting a job that requires both of these products, you should enter the following in your job command file:

```
requirements = (feature == abc) && (feature == xyz)
```

Note: One optional way to run dynamic simultaneous multithreading (SMT) is to define a feature on all machines. SMT is only supported on POWER7 processor-based systems.

Example: When submitting a job that requires the SMT function, first specify **smt = yes** in the job command file (or select a class which had **smt = yes** defined). Next, specify **job_type = parallel** and **node_usage = not_shared** and last, enter the following in the job command file:

```
requirements = (Feature == smt)
```

file_limit
Specifies the hard limit, soft limit, or both limits for the size of a file that a job can create.

Syntax:

```
file_limit = hardlimit,softlimit
```

For additional information about limit keywords, see the following topics:
- "Syntax for limit keywords" on page 295
- "Using limit keywords" on page 94

imm_send_buffers
Requests a number of immediate send buffers for each window allocated for each protocol instance of the job.

Syntax:

```
imm_send_buffers=number
```

The value of the immediate send buffers must be greater than or equal to zero. The value of the immediate send buffers is inherited from all the protocol instances of the job step unless the individual protocol instances are specified with their own immediate send buffers. If the job is sharing nodes with other jobs, then exactly *number* immediate send buffers are allocated to each window assigned to each protocol instance of the job. If the job is not sharing the nodes with other jobs then at least *number* immediate send buffers are allocated to each window assigned to each protocol instance of the job. Additional immediate send buffers can be distributed evenly to the windows assigned to the job if the nodes are not shared with other jobs.

Default value: The default value varies depending on whether the job shares the nodes with other jobs or not. If the job is not sharing the nodes with other jobs, then all the immediate send buffers are allocated to all the protocol instances of the job proportionally. If the job is sharing the nodes with other jobs, then one immediate send buffer is assigned for each window assigned to the job.

inbound_hosts
Specifies a blank-delimited list of hostnames that define the machines configured for inbound connections from other clusters.

Syntax:

```
inbound_hosts = hostname[(cluster_name)] ...
```

Where *hostname* specifies a machine configured for inbound connections from other clusters and *cluster_name* can be used to specify a specific cluster if the host is not connected to all clusters in the multicluster. These hostnames must be fully qualified with domain names if the machines exist in a different domain. This keyword is required in a multicluster environment.

Note: The same machine can be defined as both an inbound host and an outbound host.

inbound_schedd_port
Specifies the port number to use to connect to the Schedd for inbound transactions to this cluster.

Syntax:

```
inbound_schedd_port = port_number
```

Where *port_number* is a positive integer which specifies the port number used to connect to the Schedd for inbound transactions to this cluster.

Default: The default port is 9605.

include_bg

Specifies that jobs using this class will run only on the list of specified Blue Gene resources.

Syntax:

```
include_bg = list
```

Where *list* is a blank-delimited list of Blue Gene resources that can be used by jobs in this class.

An item on the *list* can be the name of a midplane (for example, R00–M0), the name of a rack (for example, R00), or the name of a row of racks (for example, R0). Other types of Blue Gene resources are not subject to any restrictions by the **include_bg** or **exclude_bg** keyword.

If both **exclude_bg** and **include_bg** are specified, **exclude_bg** takes precedence.

Default: The default is that all Blue Gene resources are included.

Examples:

In a BG/Q system, if job class ClassA has:

```
include_bg = R00
```

then jobs in ClassA can use all midplanes in rack R00 (R00–M0 and R00–M1).

include_classes

include_classes can be specified within a cluster stanza.

Specifies a blank-delimited list of one or more job classes that will accept remote jobs within the cluster.

Syntax:

```
include_classes = class_name[(cluster_name)] ...
```

Where *class_name* specifies a class to be included and *cluster_name* can be used to specify that remote jobs from *cluster_name* will be included but any other jobs submitted under *class_name* from other clusters will not be allowed.

Do not specify a list of **exclude_classes** and **include_classes**. Only one of these can be used within any cluster stanza. **exclude_classes** takes precedence over **include_classes** if both are specified.

Default: The default is that all classes are included.

include_groups

include_groups can be specified within a class stanza and a cluster stanza.

Class stanza:

When used within a class stanza, **include_groups** specifies a list of group names identifying those who can submit jobs of a particular class.

Syntax:

```
include_groups = list
```

Where *list* is a blank-delimited list of groups who are allowed to submit jobs of *class name*.

This list can contain individual user names. To allow a list of users to be included with the list of group names, add a plus sign (+) to each user name that you add to the list. LoadLeveler treats these names as implicit groups.

For example, to add user **mike** to a list of group names, specify:

```
exclude_groups = prod +mike
```

If the string **+mike** is also the actual name of a group stanza, LoadLeveler treats this name as a group, not an implicit group. In this case, LoadLeveler will not allow user **mike** to submit jobs to this class unless the user is a member of the **prod** or **+mike** group.

If this keyword is specified, this list limits groups and users of that class to those on the list.

Do not specify both a list of included groups and a list of excluded groups. Only one of these may be used for any class stanza. **exclude_groups** takes precedence over **include_groups** if both are specified.

Default: The default is that all groups are included.

Cluster stanza:

When used within a cluster stanza, **include_groups** specifies a blank-delimited list of one or more groups that will accept remote jobs within the cluster.

Syntax:

```
include_groups = group_name[(cluster_name)] ...
```

Where *group_name* specifies a group that is allowed to submit remote jobs and *cluster_name* can be used to specify that remote jobs from *cluster_name* submitted under *group_name* will be included but any other jobs submitted under *group_name* from other clusters will not be allowed.

Do not specify a list of **exclude_groups** and **include_groups**. Only one of these may be used within any cluster stanza. **exclude_groups** takes precedence over **include_groups** if both are specified.

Default: The default is that all groups are included.

include_users

include_users may be specified within a class, group, and cluster stanza.

Class or group stanza:

When used within a class or group stanza **include_users** specifies a list of user names identifying those who can submit jobs of a particular class or who are members of the group.

Syntax:

```
include_users = list
```

The definition of this keyword varies slightly, depending on the type of administration file stanza in which the keyword appears:

- In a class stanza: *list* is a blank-delimited list of users who are permitted to submit jobs of *class_name*.
- In a group stanza: *list* is a blank-delimited list of users who belong to the group.

Do not specify both a list of included users and a list of excluded users. Only one of these may be used for any class or group. **exclude_users** takes

precedence over **include_users** if both are specified. In a class stanza, users in user substanzas are also permitted to submit jobs of *class_name*, even if those users are not in the **include_users** list.

Default: The default is that all users are included.

Cluster stanza:

When used within a cluster stanza, **include_users** specifies a blank-delimited list of one or more users who can submit jobs to the cluster.

Syntax:

```
include_users = user_name[(cluster_name)] ...
```

Where *user_name* specifies a user that is allowed to submit remote jobs and *cluster_name* can be used to specify that remote jobs from *cluster_name* submitted under the *user_name* will be included but any other jobs submitted under that *user_name* from other clusters will not be allowed.

Do not specify a list of **exclude_users** and **include_users**. Only one of these may be used within any cluster stanza. **exclude_users** takes precedence over **include_users** if both are specified.

Default: The default is that all users are included.

island

Specifies the name of the island to which the machine belongs.

Syntax:

```
island = name
```

Where *name* is the name of the island. The **island** keyword can be specified in the machine stanza or the machine_group stanza. When used in a machine_group stanza, all machines belonging to that machine group belong to the specified island. The **island** keyword cannot be overridden in a machine substanza within a machine_group stanza. If the **island** keyword is not specified for a machine, then the machine does not belong to any island. A machine which does not belong to any island will not be considered by the LoadLeveler scheduler for job steps requiring the **node_topology = island** keyword.

Default value: No default value is set.

job_cpu_limit

Specifies the hard limit, soft limit, or both limits for the total amount of CPU time that all tasks of an individual job step can use per machine.

Syntax:

```
job_cpu_limit = hardlimit,softlimit
```

Example:

```
job_cpu_limit = 10000
```

For more information on this keyword, see:

- JOB_LIMIT_POLICY keyword
- For additional information about limit keywords, see the following topics:
  - "Syntax for limit keywords" on page 295
  - "Using limit keywords" on page 94

local

Specifies the scope of the cluster definition.

Syntax:

```
local = true| false
```

This keyword is required in the local cluster's administration file in a multicluster environment.

Default: false

locks_limit

Specifies the hard limit, soft limit, or both for the file locks to be used by each process of the submitted job.

Syntax:

```
locks_limit = hardlimit,softlimit
```

Examples:

```
locks_limit = 125621            # hardlimit = 125621
```

For additional information about limit keywords, see the following topics:
- "Syntax for limit keywords" on page 295
- "Using limit keywords" on page 94

machine_list

Specifies a list of machines that belong to the **machine_group**.

Syntax:

The syntax of machine_list, which consists of a list of expressions separated by comma, is modeled after the xCAT **noderange** concept. machine_list will support a subset of the xCAT syntaxes. The supported usages are:

- A single machine
  - x330n01, pccluster21, c197blade1b04
- '-' or ':' to represent a range of machines
  - Only appended numbers can be expanded, and paddings can be properly added
  - Only one range of numbers can be specified within each set of enclosing brackets
  - x330n01-x330n04 (x330n01, x330n02, x330n03, x330n04)
  - x330n01-x330n123 (x330n01, x330n02, ...x330n10... x330n99, x330n100, ... x330n123)
  - pccluster[1-1000] (pccluster1, pccluster2...pccluter1000)
  - c197blade1b[01:99] (c197blade1b01, c197blade1b02...c197blade1b99)
  - c250f08c[06-12]ap[01-08] (c250f08c06ap01, c250f08c06ap02, ..., c250f08c07ap01, c250f08c07ap02, ... c250f08c12ap08)
- '+' to represent an incremented range of machines
  - Only appended numbers can be expanded, and paddings can be properly added
  - x330n01+3 (x330n01, x330n02, x330n03, x330n04)
  - pccluster1+999 (pccluster1, pccluster2...pccluter1000)
  - c197blade1b01+98 (c197blade1b01, c197blade1b02...c197blade1b99)
- '-' serving as a prefixed exclusion operator to represent the range of machines to be excluded from the result
  - '-' as exclusion operator has precedence over other expressions
  - -x330n02, x330n01+3
  - pccluster1-1000, -pccluster21-34

The following usages are supported by xCAT, but are not supported in LoadLeveler:

- '@' because there is only one machine_list in a stanza
- '^' as file operator
- Multiple expansion
- Regular expression is not supported
- Suffix
- "node" as the default prefix

machine_mode
Specifies the type of jobs this machine can run.

Syntax:

```
machine_mode = batch | interactive | general
```

Where:

batch Specifies this machine can run only batch jobs.

interactive
Specifies this machine can run only interactive jobs. Only POE is currently enabled to run interactively.

general
Specifies this machine can run both batch jobs and interactive jobs.

Default: general

master_node_exclusive
Specifies whether or not this machine is used only as a master node.

Syntax:

```
master_node_exclusive = true| false
```

Where **true** specifies that the machine accepts only jobs (serial or parallel) submitted to classes that have **master_node_requirement** set to **true**. If the job type is parallel, only the master task is run on a machine with **master_node_exclusive** set to **true**.

Default: false

master_node_requirement
Specifies whether or not parallel jobs in this class require the master node feature.

Syntax:

```
master_node_requirement = true|false
```

Where **true** specifies that parallel jobs do require the master node feature. For these jobs, LoadLeveler allocates the first node (called the "master") on a machine having the **master_node_exclusive = true** setting in its machine stanza. If most or all of your parallel jobs require this feature, you should consider placing the statement **master_node_requirement = true** in your default class stanza. Then, for classes that do not require this feature, you can use the statement **master_node_requirement = false** in their class stanzas to override the default setting. One machine per class should have the **true** setting; if more than one machine has this setting, normal scheduling selection is performed.

Default: false

max_jobs_scheduled
Specifies the maximum number of job steps that this machine can run.

Syntax:

```
max_jobs_scheduled = number
```

Where *number* is the maximum number of jobs submitted from this scheduling (Schedd) machine that can run (or start running) in the LoadLeveler cluster at one time. If *number* of jobs are already running, no other jobs submitted from this machine will run, even if resources are available in the LoadLeveler cluster. When one of the running jobs completes, any waiting jobs then become eligible to be run.

Default: The default is -1, which means there is no maximum.

max_node
Specifies the maximum number of nodes that can be requested for a particular class or by a particular user or group for a parallel job.

Syntax:

```
max_node = number
```

Where *number* specifies the maximum number of nodes for a parallel job in a job command file using the **node** keyword.

Default: The default is -1, which means there is no limit.

max_node_resources
Specifies the maximum number of consumable resources that you can request for all tasks of a job step running on the same node.

Syntax:

```
max_node_resources = name(count) name(count) ... name(count)
```

Notes:

1. If the requirement in a job command file for the **node_resources** keyword exceeds what the administrator allows in the **max_node_resources** keyword of the corresponding class, the job will not be submitted to LoadLeveler.
2. If the **default_node_resources** specified in the class stanza exceed what is allowed by **MAX_NODE_RESOURCES**:
   - An error will be logged while reading the administration file.
   - The **max_node_resources** keyword will be ignored for the class.

Default value: No default value is set.

max_protocol_instances
Specifies the maximum number of instances on the network statement.

Syntax:

```
max_protocol_instances = number
```

Where *number* specifies the maximum value allowed on the instances keyword on the network statement for jobs submitted on this class.

Default: The default is 2.

max_reservation_duration
Specifies the maximum time, in minutes, that advance reservations made for this user or group can last.

Syntax:

```
max_reservation_duration = number of minutes
```

When the duration is defined in both the user and group stanza for a specific user, LoadLeveler uses the more restrictive of the two values to determine the maximum duration.

Default: The default is -1, which means that no limit is placed on the duration of the reservation.

For more information, see "Steps for configuring reservations in a LoadLeveler cluster" on page 128.

max_reservation_expiration

Specifies the time, in days, before a recurring reservation for a user or group must expire.

Syntax:

```
MAX_RESERVATION_EXPIRATION = number_of_days
```

The expiration date of a recurring reservation owned by this user or group can be at most **MAX_RESERVATION_EXPIRATION** days from the start time of the next occurrence of the reservation. For example, if **MAX_RESERVATION_EXPIRATION = 180** for a user, any reservation created by that user can be in the system for at most 180 days from the start of the first occurrence to the start of the last occurrence. When the maximum expiration is defined in both the user and group stanza for a specific user, LoadLeveler uses the more restrictive of the two values to determine the maximum expiration.

A value of -l means that no limit is placed on the expiration date; otherwise, the value must be a positive integer.

Default value: The default is 180.

max_reservations

Specifies the maximum number of advance reservations that this user or group can make.

Syntax:

```
max_reservations = number of reservations
```

This number includes all reservations except those in COMPLETE or CANCEL state.

Note: A recurring reservation only counts as one reservation towards the **MAX_RESERVATIONS** limit regardless of the number of times that the reservation recurs.

Table 54 summarizes the resulting behavior for various sample combinations of **max_reservations** settings in user and group stanzas.

Table 54. Sample user and group settings for the max_reservations keyword

When the user stanza value is:	And the group stanza value is:	Then the user can create this number of reservations in this group:
Not defined	Not defined	0 (zero)
2	Not defined	2 (with any group as the owning group)
Not defined	1	1
3	1	1 (the user can create more reservations in other groups)

Table 54. Sample user and group settings for the max_reservations keyword (continued)

When the user stanza value is:	And the group stanza value is:	Then the user can create this number of reservations in this group:
1	2	1
0	2	0
1	0	0 (the user can create one reservation in another group)

Default: Undefined, which means that no reservations will be authorized or disallowed. LoadLeveler considers this keyword undefined if negative values are set for it.

max_resources
Specifies the maximum number of consumable resources that you can request for a task of a job step.

Syntax:

```
max_resources = name(count) name(count) ... name(count)
```

Notes:

1. If the requirement in a job command file for the **resources** keyword or the **dstg_resources** keyword exceeds what the administrator allows in the **max_resources** keyword of the corresponding class, the job will not be submitted to LoadLeveler.
2. If the **default_resources** specified in the class stanza exceed what is allowed by **max_resources**:
   - An error will be logged while reading the administration file.
   - The **max_resources** keyword will be ignored for the class.

Default value: No default value is set.

max_starters
Specifies the maximum number of tasks that can run simultaneously on a machine. In this case, a task can be a serial job step or a parallel task. **max_starters** defines the number of initiators on the machine (the number of tasks that can be initiated from a **startd**).

Syntax:

```
max_starters = number
```

Default value: If this keyword is not specified, the default is the number of elements in the **class** statement.

For more information related to using this keyword, see "Specifying how many jobs a machine can run" on page 60.

max_top_dogs
Specifies the maximum total number of top dogs that the central manager daemon will allocate per class.

Syntax:

```
max_top_dogs = number
```

where *number* is any positive integer.

Default: The default value for this keyword is 1 for each class, unless a default is specified in the default class stanza

max_total_tasks
Specifies the maximum number of tasks that the BACKFILL scheduler allows a user, group, or class to run at any given time.

Syntax:

```
max_total_tasks = number
```

where *number* is -1, 0, or any positive integer.

Note: This keyword can be specified in a user substanza within a class when the BACKFILL scheduler is in use. This limits the number of tasks that user can run in that class at any given time.

Default: The default value for this keyword is -1, which allows an unlimited number of tasks.

maxidle
Specifies the maximum number of idle job steps this user or group can have simultaneously.

Syntax:

```
maxidle = number
```

Where *number* is the maximum number of idle jobs either this user or this group can have in queue, depending on whether this keyword appears in a user or group stanza. That is, *number* is the maximum number of jobs which the negotiator will consider for dispatch for the user or group. Jobs above this maximum are placed in the NotQueued state. This action prevents one of the following situations:

- Individual users from dominating the number of jobs that are either running or are being considered to run.
- Groups from flooding the job queue.

Notes:

1. This keyword can be specified in a user substanza within a class.
2. For the purposes of enforcing the number of idle job steps this user or group can have in queue, a job step is considered idle even if **llq** reports the state as Pending or Starting.

Default: If the user or group stanza does not specify **maxidle** or if there is no user or group stanza at all, the maximum number of jobs that can be simultaneously in queue for the user or group is defined in the default stanza. If no value is found, or the limit found is -1, then no limit is placed on the number of jobs that can be simultaneously idle for the user or group.

For more information, see "Controlling the mix of idle and running jobs" on page 413.

maxjobs
Specifies the maximum number of job steps this user, class, or group can have running simultaneously.

Syntax:

```
maxjobs = number
```

Note: This keyword can be specified in a user substanza within a class when the BACKFILL scheduler is in use.

Default: If the stanza does not specify **maxjobs**, or if there is no class, user, or group stanza at all, the maximum jobs is defined in the default stanza. The default is -1.

For more information, see "Controlling the mix of idle and running jobs" on page 413.

maxqueued
Specifies the maximum number of job steps a single group or user can have queued at the same time.

Syntax:

```
maxqueued = number
```

Where *number* is the maximum number of jobs allowed in the queue for this user or group, depending on whether this keyword appears in a user or group stanza. This is the maximum number of jobs which can be either running or being considered to be dispatched by the negotiator for that user or group. Jobs above this maximum are placed in the NotQueued state. This action prevents one of the following situations:

- Individual users from dominating the number of jobs that are either running or are being considered to run.
- Groups from flooding the job queue.

Note: This keyword can be specified in a user substanza within a class.

Default: If the user or group stanza does not specify **maxqueued** or if there is no user or group stanza at all, the maximum number of jobs that can be simultaneously in queue for the user or group is defined in the default stanza. If no value is found, or the limit found is -1, then no limit is placed on the number of jobs that can be simultaneously idle for the user or group. Regardless of this limit, there is no limit to the number of jobs a user or group can submit.

For more information, see "Controlling the mix of idle and running jobs" on page 413.

memlock_limit
Specifies the hard limit, soft limit, or both for the memory that can be locked by each process of the submitted job.

Syntax:

```
memlock_limit = hardlimit,softlimit
```

Examples:

```
memlock_limit = 125621          # hardlimit = 125621 bytes
memlock_limit = 5621kb          # hardlimit = 5621 kilobytes
memlock_limit = 2mb             # hardlimit = 2 megabytes
memlock_limit = 2.5mw           # hardlimit = 2.5 megawords
memlock_limit = unlimited       # hardlimit = 9,223,372,036,854,775,807 \
                                  bytes (X'7FFFFFFFFFFFFFFF')
memlock_limit = rlim_infinity   # hardlimit = 9,223,372,036,854,775,807 \
                                  bytes (X'7FFFFFFFFFFFFFFF')
```

For additional information about limit keywords, see the following topics:
- "Syntax for limit keywords" on page 295
- "Using limit keywords" on page 94

multicluster_security
Specifies a security mechanism to use for authentication and authorization of intercluster communications.

Syntax:

```
multicluster_security = SSL
```

The only valid specification for this keyword is **SSL**. When **SSL** is specified, LoadLeveler uses the OpenSSL library to provide secure intercluster transactions. If this keyword is omitted or left blank and the **MACHINE_AUTHENTICATE** in the configuration file is set to **true**, then LoadLeveler will accept intercluster transactions only from machines listed as **inbound_hosts** or **outbound_hosts** in the administration file. Otherwise, intercluster transactions are accepted from any machine.

For more information, see "Steps for securing communications within a LoadLeveler multicluster" on page 153.

name_server
Specifies a list of name servers used for a machine.

Syntax:

```
name_server = list
```

Where *list* is a blank-delimited list of character strings that is used to specify which nameservers are used for the machine. Valid strings are DNS, NIS, and LOCAL. LoadLeveler uses the list to determine when to append a DNS domain name for machine names specified in LoadLeveler commands.

If DNS is specified alone, LoadLeveler will always append the DNS domain name to machine names specified in LoadLeveler commands. If NIS or LOCAL is specified, LoadLeveler will never append a DNS domain name to machine names specified in LoadLeveler commands.

Note: The **name_server** keyword is a cluster-wide keyword that can only be specified for the default machine or default machine group stanza.

nice
Increments the *nice* value of a job.

Syntax:

```
nice = value
```

Where *value* is the amount by which the current UNIX *nice* value is incremented. The *nice* value is one factor in a job's run priority. The lower the number, the higher the run priority. If two jobs are running on a machine, the *nice* value determines the percentage of the CPU allocated to each job.

The default value is 0. If the administrator has decided to enforce consumable resources, the *nice* value will only adjust priorities of processes within the same WLM class. Because LoadLeveler defines a single class for every job step, the *nice* value has no effect.

For more information, consult the appropriate UNIX documentation.

nofile_limit
Specifies the hard limit, soft limit, or both for the number of open file descriptors that can be used by each process of the submitted job.

Syntax:

```
nofile_limit = hardlimit,softlimit
```

Examples:

```
nofile_limit = 1000             # hardlimit = 1000
```

For additional information about limit keywords, see the following topics:

- "Syntax for limit keywords" on page 295
- "Using limit keywords" on page 94

nproc_limit

Specifies the hard limit, soft limit, or both for the number of processes that can be created for the real user ID of the submitted job.

Syntax:

```
nproc_limit = hardlimit,softlimit
```

Examples:

```
nproc_limit = 256              # hardlimit = 256
```

For additional information about limit keywords, see the following topics:
- "Syntax for limit keywords" on page 295
- "Using limit keywords" on page 94

outbound_hosts

Blank-delimited list of hostnames that define the machines configured for outbound connections to other clusters.

Syntax:

```
outbound_hosts = hostname[(cluster_name)] ...
```

Where *hostname* specifies a machine configured for outbound connections to other clusters and *cluster_name* can be used to specify a specific cluster if the host is not connected to all clusters in the multicluster. These hostnames must be fully qualified with domain names if the machines exist in a different domain. This keyword is required in a multicluster environment.

Note: The same machine can be defined as both an outbound host and an inbound host.

pool_list

Specifies a list of pool numbers to which the machine belongs. Do not use negative numbers in a machine pool_list.

Syntax:

```
pool_list = pool_numbers
```

Where *pool_numbers* is a blank-delimited list of non-negative numbers identifying pools to which the machine belongs. These numbers can be any positive integers including zero.

power_management_policy

Syntax:

```
power_management_policy = start_time;duration | off
```

Where:

off
Disables the site energy policy.

start_time
Is the same format used by crontab.

duration
Is the number of minutes the machine will be in standby state. Set the duration and the time between durations to a value more than 10 minutes.

Example 1:

In this example, the machine will go in standby state from 0 o'clock every day for 5 hours:

```
power_management_policy=00 00 * * *; 18000
```

The machine will stay in standby state at least 5 hours if there is no job assigned to it.

Example 2:

In this example, the node will be in standby state when there is no job running. The node will wake up when a job is dispatched to run on it, or when the **llrchgmstat** command is issued to wake it up:

```
power_management_policy=* * * * *
```

For more information about the **llrchgmstat** command, see *LoadLeveler: Resource Manager*.

Example 3:

In this example, node c197blade4b06 will use the site policy that is configured in machine group mgroup1 and node c197blade4b05 will use the site policy that is set in the machine stanza:

```
c197blade4b05.ppd.pok.ibm.com: {
        type = machine
        central_manager = true
     schedd_host = true
     class = No_Class(128)
     power_management_policy = 00 03 1 3 *;1800
}
mgroup1: {
        type = machine_group
     schedd_runs_here = false
     startd_runs_here = true
     max_starters = 128
     class = No_Class(128)
     machine_mode = general
     machine_list =  c197blade4b06.ppd.pok.ibm.com
     power_management_policy = 00 03 1 5 *;1200
}
```

prestarted_starters

Specifies how many prestarted starter processes LoadLeveler will be maintained on an execution node to manage jobs when they arrive. The startd daemon starts the number of starter processes specified by this keyword.

Syntax:

```
prestarted_starters = number
```

number must be less than or equal to the value specified through the **max_starters** keyword. If the value of **prestarted_starters** specified is greater then **max_starters**, LoadLeveler records a warning message in the startd log and assigns **prestarted_starters** the same value as **max_starters**.

If the value **prestarted_starters** is zero, no starter processes will be started before jobs arrive on the execution node.

Default value: The default is 1.

priority

Identifies the priority of the appropriate user, class, or group.

Syntax:

```
priority = number
```

Where *number* is a integer that specifies the priority for jobs either in this class, or submitted by this user or group, depending on whether this keyword appears in a class, user, or group stanza, respectively.

The number specified for priority is referenced as either **ClassSysprio**, **UserSysprio**, or **GroupSysprio** in the configuration file. You can use **ClassSysprio**, **UserSysprio**, or **GroupSysprio** when assigning job priorities. If the variable **ClassSysprio**, **UserSysprio**, or **GroupSysprio** does not appear in the **SYSPRIO** expression in the configuration file, then the priority specified in the administration file is ignored. See "LoadLeveler variables" on page 286 for more information about the **ClassSysprio**, **UserSysprio**, or **GroupSysprio** keywords.

Default: The default is 0.

region
Defines the region to which the machine or **machine_group** belongs. The machine cannot be defined as the primary region manager or alternate region manager of another region. Substanzas cannot use this keyword. All machines within a **machine_group** can belong to only one region.

Syntax:

```
region = region_name
```

Default: No default value is set.

reservation_type
Specifies the type of reservation users can use.

Syntax:

```
reservation_type = all | none | flexible
```

Default: The default is **all**.

resources
Specifies quantities of the consumable resources initially available on the machine.

Syntax:

```
resources = name(count) name(count) ... name(count)
```

Where *name(count)* is an administrator-defined name and count, or could also be **ConsumableCpus***(count)*, **ConsumableMemory***(count units)*, **ConsumableVirtualMemory***(count units)*, or **ConsumableLargePageMemory***(count units)*.

The **ConsumableMemory**, **ConsumableVirtualMemory**, and **ConsumableLargePageMemory** can be specified with both a count and units. The **ConsumableMemory** or **ConsumableVirtualMemory** specified resource count must be an integer greater than zero. The **ConsumableLargePageMemory** specified resource count must be an integer greater than or equal to zero. The allowable units are those normally used with LoadLeveler data limits:

```
b bytes
w words
kb kilobytes (2**10 bytes)
kw kilowords (2**12 bytes)
mb megabytes (2**20 bytes)
mw megawords (2**22 bytes)
gb gigabytes (2**30 bytes)
gw gigawords (2**32 bytes)
tb terabytes (2**40 bytes)
```

```
tw terawords (2**42 bytes)
pb petabytes (2**50 bytes)
pw petawords (2**52 bytes)
eb exabytes (2**60 bytes)
ew exawords (2**62 bytes)
```

The **ConsumableMemory**, **ConsumableVirtualMemory**, and **ConsumableLargePageMemory** values are stored in MB (megabytes) and rounded up. For **ConsumableMemory** or **ConsumableVirtualMemory**, the smallest amount that you can request is one megabyte. If no units are specified, then megabytes are assumed. Resources defined here that are not in the **SCHEDULE_BY_RESOURCES** list in the global configuration file will not effect the scheduling of the job.

For the **ConsumableCpus** resource, a value of **all** can be specified instead of count. This indicates that the CPU resource value will be obtained from the Startd daemons. However, these resources will not be available for scheduling until the first **Startd** update.

A list within < > angle brackets indicates a list of CPU IDs. Only CPUs with logical IDs specified in the list will be considered available for LoadLeveler jobs. The following example specifies a list of CPUs:

```
resources = ConsumableCpus< 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 >
```

CPU IDs can also be specified using a list of ranges:

```
resources = ConsumableCpus< 0-6 10-16 >
```

Jobs requesting processor affinity with the **task_affinity** keyword in the job command file will only run on machines where the resource statement contains the **ConsumableCpus** keyword.

The logical IDs of the CPUs available on a machine can be found issuing the `bindprocessor -q` command.

Default: No default value is set.

restart

Specifies whether LoadLeveler considers a job to be restartable.

Syntax:

```
restart = yes | no
```

Default: yes

rss_limit

Specifies the hard limit, soft limit, or both limits for the resident set size for a job.

Syntax:

```
rss_limit = hardlimit,softlimit
```

For additional information about limit keywords, see the following topics:

- "Syntax for limit keywords" on page 295
- "Using limit keywords" on page 94

schedd_fenced

Specifies whether or not the central manager is to ignore connections from the Schedd daemon running on this machine.

Syntax:

```
schedd_fenced = true | false
```

Where **true** specifies that the central manager ignores connections from the Schedd daemon running on this machine. Use the **true** setting together with the **llmovespool** command when you want to attempt to recover resources lost when a node running the Schedd daemon fails. A **true** setting prevents conflicts when you use the **llmovespool** command to move jobs from one Schedd machine to another. For more information, see "How do I recover resources allocated by a Schedd machine?" on page 401.

Default: false

schedd_host
Specifies whether or not this machine is used to help submit-only machines access LoadLeveler hosts that run LoadLeveler jobs.

Syntax:

```
schedd_host = true | false
```

When **true** this keyword specifies that if a Schedd is running on a machine that it will serve as a public scheduling machine. A public scheduling machine accepts job submissions from other machines in the LoadLeveler cluster. Jobs are submitted to a public scheduling machine if:

- The submission occurs on a machine which does not run the Schedd daemon. These include submit-only machines and machines which are configured to run other LoadLeveler daemons but not the Schedd daemon.
- The submission occurs on a machine which runs the Schedd daemon but is configured to submit jobs to a public scheduling machine by having the **SCHEDD_SUBMIT_AFFINITY** keyword set to **false** in the global or local configuration file.

This keyword does not configure LoadLeveler to run the Schedd daemon on a node. Use the administration keyword **schedd_runs_here** to run the Schedd daemon on a node.

Default: false

schedd_runs_here
Specifies whether the Schedd daemon runs on the host. If you do not want to run the Schedd daemon, specify **false**.

This keyword does not designate a machine as a public scheduling machine. Unless configured as a public scheduling machine, a machine configured to run the Schedd daemon will only accept job submissions from the same machine running the Schedd daemon. A public scheduling machine accepts job submissions from other machines in the LoadLeveler cluster. To configure a machine as a public scheduling machine, see the **schedd_host** keyword description in "Administration keyword descriptions" on page 298.

Syntax:

```
schedd_runs_here = true | false
```

Default value: true

secure_schedd_port
Specifies the port number to use to connect to the Schedd for secure inbound transactions to this cluster.

Syntax:

```
secure_schedd_port = port_number
```

Where *port_number* is a positive integer that specifies the port number used to connect to the Schedd for secure inbound transactions to this cluster. This port is only used if the **multicluster_security** keyword is set to **SSL**. The secure Schedd port should be different from the normal Schedd port.

Default: 9607

smt

Indicates the required simultaneous multithreading (SMT) state for all job steps assigned to the class.

Syntax:

```
smt = yes | no | as_is
```

Where:

yes The job step requires SMT to be enabled.

no The job step requires SMT to be disabled.

as_is The SMT state will not be changed.

Note: You cannot use the **smt** and **rset** keywords together if **smt** is set to either **yes** or **no**. Using these keywords together will cause your job to fail.

Default value: as_is

Examples:

```
smt = yes
```

speed

Specifies the weight associated with the machine for scheduling purposes.

Syntax:

```
speed = number
```

Where *number* is a floating point number that is used for machine scheduling purposes in the **MACHPRIO** expression. For more information on machine scheduling and the MACHPRIO expression, see "Setting negotiator characteristics and policies" on page 47. In addition, the **speed** keyword is also used to define the weight associated with the machine. This weight is used when gathering accounting data on a machine basis.

To distinguish speed among different machines, you must include this value in the local configuration file. For information on how the **speed** keyword can be used to schedule machines, refer to "Setting negotiator characteristics and policies" on page 47.

Default: The default is 1.0.

ssl_cipher_list

Specifies a cipher list defining what encryption methods are available to OpenSSL when securing multicluster connections.

Syntax:

```
ssl_cipher_list = cipher_list
```

Where *cipher_list* is a valid cipher list as documented by the OpenSSL **ciphers** command.

Default: This keyword will default to the "ALL:eNULL:!aNULL" string.

stack_limit
Specifies the hard limit, soft limit, or both limits for the size of a stack.

Syntax:

```
stack_limit = hardlimit,softlimit
```

For additional information about limit keywords, see the following topics:
- "Syntax for limit keywords" on page 295
- "Using limit keywords" on page 94

startd_runs_here = true | false
Specifies whether the startd daemon runs on the host. If you do not want to run the startd daemon, specify **false**.

Syntax:

```
startd_runs_here  = true | false
```

Default value: true

striping_with_minimum_networks
Specifies whether or not nodes which have more than half of their networks in READY state are considered for **sn_all** jobs. This makes certain that at least one network is UP and in READY state between any two nodes assigned for the job.

Syntax:

```
striping_with_minimum_networks = true| false
```

When set to true, then the nodes which have more than half the minimum number of networks in the READY state are considered for **sn_all** jobs. If set to **false**, then only nodes which have all networks in the READY state are considered for **sn_all** jobs.

Default: false

submit_only
Specifies whether or not this machine is a submit-only machine.

Syntax:

```
submit_only = true| false
```

Where **true** designates this as a submit-only machine. If you set this keyword to **true**, in the administration file set **central_manager** and **schedd_host** to **false**.

Default: false

total_tasks
Specifies the maximum number of tasks that can be requested for a particular class or by a particular user or group for a parallel job.

Syntax:

```
total_tasks = number
```

Where *number* specifies the maximum number of tasks for a parallel job in a job command file using the **total_tasks** keyword.

Default: The default is -1, which means there is no limit.

type
Identifies the type of stanza in the administration file.

Syntax:

```
type = stanza_type
```

Where *stanza_type* is one of the following:
- Class
- Cluster
- Group
- Machine
- Region
- User

Default: No default value is set.

wall_clock_limit
Specifies the hard limit, soft limit, or both limits for the amount of elapsed time for which a job can run.

Syntax:

```
wall_clock_limit = hardlimit,softlimit
```

Note that LoadLeveler uses the time the negotiator daemon dispatches the job as the start time of the job. When a job is checkpointed, vacated, and then restarted, the **wall_clock_limit** is not adjusted to account for the amount of time that elapsed before the checkpoint occurred.

If you are running the BACKFILL scheduler, you must set a wall clock limit either in the job command file or in a class stanza (for the class associated with the job you submit). LoadLeveler administrators should consider setting a default wall clock limit in a default class stanza. For more information on setting a wall clock limit when using the BACKFILL scheduler, see "Choosing a scheduler" on page 46.

For additional information about limit keywords, see the following topics:
- "Syntax for limit keywords" on page 295
- "Using limit keywords" on page 94

Chapter 12. Job command file reference

A LoadLeveler job consists of one or more job steps, each of which is defined in a single job command file. A job command file specifies the name of the job, as well as the job steps that you want to submit, and can contain other LoadLeveler statements.

Table 55 lists the job command file subtasks:

Table 55. Job command file subtasks

Subtask	Associated information (see . . .)
To find out how to work with a job command file	Chapter 7, "Building and submitting jobs," on page 171
To learn how to correctly specify the contents of a job command file	• "Job command file syntax" • "Job command file keyword descriptions" on page 335

Job command file syntax

There are general rules that apply to job command files.

- Keyword statements begin with # @. There can be any number of blanks between the # and the @.
- Comments begin with #. Any line whose first non-blank character is a pound sign (#) and is not a LoadLeveler keyword statement is regarded as a comment.
- Statement components are separated by blanks. You can use blanks before or after other delimiters to improve readability but they are not required if another delimiter is used.
- The back-slash (\) is the line continuation character. Note that the continued line must not begin with # @. If your job command file is the script to be executed, you must start the continued line with a #. See Example 2 and Example 3 in topic "Examples: Job command files" on page 173 for examples that use the back-slash for line continuation.
- Keywords are *not* case sensitive. This means you can enter them in lower case, upper case, or mixed case.

Serial job command file

The serial job command file is run from the current working directory.

The following is an example of a simple serial job command file which is run from the current working directory. The job command file reads the input file, **longjob.in1**, from the current working directory and writes standard output and standard error files, **longjob.out1** and **longjob.err1**, respectively, to the current working directory.

```
#   The name of this job command file is file.cmd.
#   The input file is longjob.in1 and the error file is
#   longjob.err1. The queue statement marks the end of
#   the job step.
#
# @ executable = longjob
```

```
# @ input = longjob.in1
# @ output = longjob.out1
# @ error = longjob.err1
# @ queue
```

Parallel job command file

In addition to building job command files to submit serial jobs, you can also build job command files to submit parallel jobs.

Before constructing parallel job command files, consult your LoadLeveler system administrator to see if your installation is configured for parallel batch job submission.

For more information on submitting parallel jobs, see "Working with parallel jobs" on page 184.

Syntax for limit keywords

There is a syntax for limit keywords. See "Syntax for limit keywords" on page 295 for additional information about limit keywords.

64-bit support for job command file keywords

Users can assign 64-bit integer values to selected keywords in the job command file.

System resource limits, with the exception of CPU limits, are treated by LoadLeveler daemons and commands as 64-bit limits.

Table 56 describes 64-bit support for specific job command file keywords.

Table 56. Notes on 64-bit support for job command file keywords

Keyword name	Notes
as_limit	64-bit integer values can be assigned to these limits. Fractional specifications are allowed and will be converted to 64-bit integer values. Refer to the allowable units for these limits listed under the "Syntax for limit keywords" on page 295 topic.
ckpt_time_ limit	Not supported. The hard and soft time limits associated with this keyword are 32-bit integers. If a value that cannot be contained in a 32-bit integer is assigned to this limit, the value will be truncated to either 2147483647 or -2147483648.
core_limit	64-bit integer values may be assigned to this limit. Fractional specifications are allowed and will be converted to 64-bit integer values. Refer to the allowable units for these limits listed under "Syntax for limit keywords" on page 295.
cpu_limit	Not supported. The hard and soft time limits associated with this keyword are 32-bit integers. If a value that cannot be contained in a 32-bit integer is assigned to this limit, the value will be truncated to either 2147483647 or -2147483648.
data_limit	64-bit integer values may be assigned to these limits. Fractional specifications are allowed and will be converted to 64-bit integer values. Refer to the allowable units for these limits listed under "Syntax for limit keywords" on page 295.
file_limit	
image_size	64-bit integer values may be assigned to this keyword. Fractional and unit specifications are not allowed. The default unit of **image_size** is kb. **Example:** `image_size = 12345678901`
job_cpu_limit	Not supported. The hard and soft time limits associated with this keyword are 32-bit integers. If a value that cannot be contained in a 32-bit integer is assigned to this limit, the value will be truncated to either 2147483647 or -2147483648.

Table 56. Notes on 64-bit support for job command file keywords (continued)

Keyword name	Notes
locks_limit	64-bit integer values can be assigned to these limits. Fractional specifications are allowed and will be converted to 64-bit integer values. Refer to the allowable units for these limits listed under the "Syntax for limit keywords" on page 295 topic.
memlock_limit	See notes for **as_limit**.
nofile_limit	See notes for **locks_limit**.
nproc_limit	See notes for **locks_limit**.

preferences
requirements
	64-bit integer values may be associated with the LoadLeveler variables "Memory" and "Disk" in the expressions assigned to these keywords. Fractional and unit specifications are not allowed.
	Examples:
	`requirements = (Arch == "R6000") && (Disk > 500000000) && (Memory > 6000000000)`
	`preferences = (Disk > 6000000000) && (Memory > 9000000000)`

resources
node_resources
	Consumable resources associated with the **resources** keyword may be assigned 64-bit integer values. Fractional specifications are not allowed. Unit specifications are valid only when specifying the values of the predefined ConsumableMemory, ConsumableVirtualMemory, and ConsumableLargePageMemory resources.
	Examples:
	`resources = spice2g6(123456789012) ConsumableMemory(10 gb)`
	`resources = ConsumableVirtualMemory(15 pb) db2_license(1)`
	`resources = ConsumableLargePageMemory(2048mb)`

rss_limit **stack_limit**	64-bit integer values may be assigned to these limits. Fractional specifications are allowed and will be converted to 64-bit integer values. Refer to the allowable units for these limits listed under "Syntax for limit keywords" on page 295.
wall_clock_ limit	Not supported. The hard and soft time limits associated with this keyword are 32-bit integers. If a value that cannot be contained in a 32-bit integer is assigned to this limit, the value will be truncated to either 2147483647 or -2147483648.

Job command file keyword descriptions

This topics contains an alphabetical list of the keywords you can use in a LoadLeveler script.

This topic also provides examples of statements that use these keywords. For most keywords, if you specify the keyword in a job step of a multistep job, its value is inherited by all proceeding job steps. Exceptions to this are noted in the keyword description.

If a blank value is used after the equal sign, it is as if no keyword was specified.

account_no

Supports centralized accounting. Allows you to specify an account number to associate with a job. This account number is stored with job resource information in local and global history files. It may also be validated before LoadLeveler allows a job to be submitted. For more information, see "Gathering job accounting data" on page 65.

Syntax:

account_no = *string*

where *string* is a text string that can consist of a combination of numbers and letters.

Default value: No default value is set.

Example: If the job accounting group charges for job time based upon the department to which you belong, your account number would be similar to:

```
account_no = dept34ca
```

adjust_wall_clock_limit

Indicates if the **wall_clock_limit** for the job will be adjusted automatically by LoadLeveler when the job runs with a lower CPU frequency.

Syntax:

```
adjust_wall_clock_limit = yes | no
```

where:

yes
Indicates that LoadLeveler will increase the wall clock limit of the job when the job runs with a lower CPU frequency. LoadLeveler will add twice the estimated performance degradation to the original wall clock limit to get the adjusted wall clock limit. For example, if the job has 5 percent performance degradation, the adjusted wall clock limit will be set to (1 + 2 * 5%) * (step wall clock limit). The job will be stopped if the job cannot be finished with the adjusted wall clock time. The adjusted wall clock limit can exceed the wall clock limit value, which is set in class stanza. When the wall clock limit (after adjustment) is longer than the value of the wall clock time in the class stanza, the adjusted wall clock limit value will be used.

no LoadLeveler will not increase the wall clock limit for the job when the job runs with a lower CPU frequency. You must increase the wall clock limit on your own when the job runs with a lower frequency.

Note: This keyword is only applicable when the energy function is enabled.

Default value: yes

arguments

Specifies the list of arguments to pass to your program when your job runs.

Syntax:

```
arguments = arg1 arg2 ...
```

Default value: No default arguments are set.

Example: If your job requires the numbers 5, 8, 9 as input, your arguments keyword would be similar to:

```
arguments = 5 8 9
```

as_limit

Specifies the hard limit, soft limit, or both limits for the size of address space that the submitted job can use. This limit is a per process limit.

Syntax:

```
as_limit = hardlimit,softlimit
```

Default value: No default value is set.

Example:

```
as_limit = ,125621
as_limit = 5621kb
as_limit = 2mb
as_limit = 2.5mw,2mb
```

For additional information about limit keywords, see the following topics:
- "Syntax for limit keywords" on page 295
- "Using limit keywords" on page 94

bg_block

Specifies the name of an existing block on the Blue Gene system in which the job is to be started.

Syntax:

bg_block = *block_name*

where *block_name* is the name identifying a block in the Blue Gene system.

This keyword is only valid for **job_type = bluegene**. The **bg_block** keyword cannot be used if any of the following keywords are specified: **bg_size**, **bg_shape**, **bg_connectivity**, and **bg_rotate**. The value of the keyword applies only to the job step in which you specify the keyword (that is, the keyword is not inherited by other job steps).

Default value: No default is set.

bg_connectivity

Specifies the type of connectivity requested for the Blue Gene block in which the job step will run. Torus or Mesh can be specified on a per dimension basis. The E dimension is always a Torus. Connectivity is only applicable for large block jobs (>= 1 midplane). If the size in a given dimension is equal to 1 or the dimension size, and connectivity specified is not Torus, the connectivity will be changed to Torus.

Syntax:

bg_connectivity = Torus | **Mesh** | **Either** | *Xa Xb Xc Xd*

where:

Xa Is equal to Torus or Mesh, specified for the A dimension.

Xb Is equal to Torus or Mesh, specified for the B dimension.

Xc Is equal to Torus or Mesh, specified for the C dimension.

Xd Is equal to Torus or Mesh, specified for the D dimension.

Torus Specifies that the admissible blocks should have Torus connectivity in all dimensions.

Mesh Specifies that the admissible blocks should have Mesh connectivity in all dimensions.

Either Specified for the entire block (not on a dimension basis) and will set the connectivity to Torus for all dimensions, if possible, otherwise, it will set the connectivity to Mesh for all dimensions.

This keyword is only valid for **job_type = bluegene**. The **bg_connectivity** keyword cannot be used if the **bg_block** keyword is specified. The value of the keyword applies only to the job step in which you specify the keyword (that is, the keyword is not inherited by other job steps).

Default value: Mesh

Examples:

```
bg_connectivity = Torus Mesh Mesh Torus
```

will set the connections per dimension as follows:

```
A: Torus
B: Mesh
C: Mesh
D: Torus
E: Torus
```

```
bg_connectivity = Mesh
```

will set the connections per dimension as follows:

```
A: Mesh
B: Mesh
C: Mesh
D: Mesh
E: Torus
```

bg_requirements

Specifies the requirements which a Blue Gene midplane in the LoadLeveler cluster must meet to run any job steps.

Syntax:

```
bg_requirements = Boolean_expression
```

The only requirement supported at this time is memory, where memory specifies the amount, in megabytes, of regular physical memory required in the C-nodes of the Blue Gene midplane where you want your job step to run.

Example 1: To require Blue Gene midplanes with 512 megabytes of physical memory in their C-nodes, enter:

```
bg_requirements = (Memory == 512)
```

Example 2: To require Blue Gene midplanes with more than 512 megabytes of physical memory in their C-nodes, enter:

```
bg_requirements = (Memory > 512)
```

This keyword is only valid for **job type= bluegene**. This keyword cannot be used if the **bg_block** keyword is specified. This keyword is not inherited by other job steps.

Default value: No default value is set.

bg_rotate

Specifies whether the scheduler should consider all possible rotations of the given shape of the job when searching for a block for the job.

Syntax:

```
bg_rotate = true | false
```

where the value **true** implies that the shape can be rotated to fit some free block and the value **false** implies that the shape will not be rotated. The default for the **bg_rotate** job command file keyword is **true**, so not specifying this keyword is equivalent to specifying **bg_rotate = true**.

Using **bg_rotate = true** will increase the likelihood of the scheduler finding a block to run the job and optimizes overall scheduling of Blue Gene resources.

Note: Be aware that this keyword should be set to **false** when using the **--mapping** option for **runjob** to specify how the job tasks are to be assigned to the allocated compute nodes.

Example 1:

Suppose for a 4x4x4x2 system, a particular job command file specifies keywords **bg_rotate = true** and **bg_shape = 3x1x2x4**. There exists the following 24 possible rotations of the shape specified:

```
1x2x3x4, 1x2x4x3, 1x3x4x2, 1x3x2x4, 1x4x2x3, 1x4x3x2
2x1x3x4, 2x1x4x3, 2x3x4x1, 2x3x1x4, 2x4x1x3, 2x4x3x1
3x1x2x4, 3x1x4x2, 3x2x4x1, 3x2x1x4, 3x4x1x2, 3x4x2x1,
4x1x2x3, 4x1x3x2, 4x2x3x1, 4x2x1x3, 4x3x1x2, 4x3x2x1
```

However, due to the machine size constraint, only the following shapes are valid:

```
1x3x4x2, 1x4x3x2, 2x3x4x1, 2x4x3x1, 3x1x4x3, 3x2x4x1
3x4x1x2, 3x4x2x1, 4x1x3x2, 4x2x3x1, 4x3x1x2, 4x3x2x1
```

Example 2:

Suppose that some other job command file specifies keywords **bg_rotate = false** and **bg_shape = 3x1x2x4**, and that there is an available block of shape 4x1x3x2. Then the scheduler will not be able to use that block for this job.

This keyword is only valid for **job type= bluegene**. The **bg_rotate** keyword cannot be used if the **bg_block** keyword is specified. The value of the **bg_rotate** keyword applies only to the job step in which you specify the keyword (that is, this keyword is not inherited by other job steps.)

In BG/Q, the connectivity is specified on a per-dimension basis and the connectivity will be rotated along with the shape. This means if the user specifies a shape of 3x1x2x4 (*AxBxCxD*) with a connectivity of Torus(A) Mesh(B) Torus(C) Mesh(D) then a rotation to 4x2x3x1 (*AxBxCxD*) would result in a connectivity of Mesh(A) Torus(B) Torus(C) Mesh(D).

Note: This keyword can only be used together with the **bg_shape** job command file keyword. If the **bg_shape** keyword is not present, then this keyword is not relevant.

Default value: The default value is **true**.

bg_shape

Specifies the requested shape of the Blue Gene job to be started in the system.

Syntax:

bg_shape = *AxBxCxD*

where: where *A*, *B*, *C*, and *D* are the number of midplanes in the A-direction, B-direction, C-direction, and D-direction, respectively, of the requested **bg_shape** of the job. This keyword is only valued for **job_type = bluegene**. The **bg_shape** keyword cannot be used if the **bg_size** or **bg_blocks** keywords are specified.

The values of *A*, *B*, *C*, and *D* must not be greater than the corresponding *A*, *B*, *C*, and *D* sizes of the Blue Gene machine, otherwise, the job will never be able to start. The maximum supported size of BG/Q is 1024 racks (32x32) or 2048 midplanes in an 8x8x8x4 configuration.

Note: In order to have the *A*, *B*, *C*, and *D* dimensions of the allocation block exactly as defined by the **bg_shape** job command file keyword, the job command file keyword **bg_rotate** needs to be set to **false**.

Default value: No default is set.

bg_size

Specifies the requested size of the Blue Gene job to be started in the system.

Syntax:

bg_size = *bg_size*

where *bg_size* is an integer indicating the size of the job in units of compute nodes. No guarantees are made as to the shape of the allocated block for a given size. The only guarantee is that the size of the allocated shape will be no smaller than the requested size and as close to the request size as possible.

This keyword is only valid for job type **bluegene**. This keyword cannot be used if the **bg_block** or **bg_shape** keyword is specified. This keyword is not inherited by other job steps.

Note: Not all values given for **bg_size** are representable. For example, consider a 2x4x2x2 Blue Gene system in units of midplanes and a requested **bg_size** of 5632 (equivalent to 11 midplanes). Because 11 is a prime number, it cannot be decomposed. Furthermore, it is greater than any one dimension of the system. In this case, a 2x3x2x1 block is allocated, because it is the smallest number of midplanes larger than the requested size.

Default value: If **bg_size**, **bg_shape**, or **bg_block** are not specified then **bg_size** defaults to the configured minimum block size. This is the value of the **BG_MIN_BLOCK_SIZE** keyword in the configuration file.

blocking

Blocking specifies that tasks be assigned to machines in multiples of a certain integer. Unlimited blocking specifies that tasks be assigned to each machine until it runs out of initiators, at which time tasks will be assigned to the machine which is next in the order of priority. If the total number of tasks are not evenly divisible by the blocking factor, the remainder of tasks are allocated to a single node.

This keyword is supported by the BACKFILL and API schedulers.

Syntax:

blocking = *integer* **|** **unlimited**

where:

integer
Specifies the blocking factor to be used. The blocking factor must be a positive integer. With a blocking factor of 4, LoadLeveler will allocate 4 tasks at a time to each machine with at least 4 initiators available. This keyword must be specified with the total_tasks keyword. **Example:**

```
blocking = 4
total_tasks = 17
```

LoadLeveler will allocate tasks to machines in an order based on the values of their MACHPRIO expressions (beginning with the highest MACHPRIO value). In cases where total_tasks is not a multiple of the blocking factor, LoadLeveler assigns the remaining number of tasks as soon as possible (even if that means assigning the remainder to a machine at the same time as it assigns another block).

unlimited
Specifies that LoadLeveler allocate as many tasks as possible to each machine, until all of the tasks have been allocated. LoadLeveler will prioritize machines based on the number of initiators each machine

currently has available. Unlimited blocking is the only means of allocating tasks to nodes that does not prioritize machines primarily by MACHPRIO expression.

Default value: No default is set, which means that no blocking is requested.

bulkxfer

Indicates whether the communication subsystem will use bulk data transfer for user space communication.

Syntax:

```
bulkxfer = yes | no
```

Default: no

For additional information about bulk data transfer, see "Using bulk data transfer" on page 180.

checkpoint

Indicates if a job is able to be checkpointed. Checkpointing a job is a way of saving the state of the job so that if the job does not complete it can be restarted from the saved state rather than starting the job from the beginning.

If you specify a value that is not valid for the **checkpoint** keyword, an error message is generated and the job is not submitted.

Syntax:

```
checkpoint = interval [(number)] | yes | no
```

where:

interval
Specifies that LoadLeveler will automatically checkpoint your program at preset intervals. The time interval can be optionally specified by providing the number of seconds for the interval with the interval option. If a value for the interval is not specified, the settings in the **MIN_CKPT_INTERVAL** and **MAX_CKPT_INTERVAL** keywords in the configuration file will be used. Because a job with a setting of **interval** is considered checkpointable, you can initiate a checkpoint using any method in addition to the automatic checkpoint. The difference between **interval** and **yes** is that **interval** enables LoadLeveler to automatically take checkpoints on the specified intervals, while the value **yes** does not enable that ability.

yes Enables a job step to be checkpointed. With this setting, a checkpoint can be initiated either under the control of an application or by a method external to the application. With a setting of **yes**, LoadLeveler will not checkpoint on the intervals specified by the **MIN_CKPT_INTERVAL** and **MAX_CKPT_INTERVAL** keywords in the configuration file. The difference between **yes** and **interval** is that **interval** enables LoadLeveler to automatically take checkpoints on the specified intervals while the value **yes** does not enable that ability.

no The step cannot be checkpointed.

Default value: no

Restriction: If a job with **checkpoint = interval** or **checkpoint = yes** is dispatched to an AIX machine that does not have MetaCluster installed, the job is rejected.

Example: If a checkpoint is initiated from within the application but checkpoints are not to be taken automatically by LoadLeveler you can use:

```
checkpoint = yes
```

Tips:

- The location of the checkpoint files will be determined by keyword values set in the job command file, the administration file, or by default values.
- The **checkpoint** keyword value can be in mixed case.

For detailed information on checkpointing, see "LoadLeveler support for checkpointing jobs" on page 135.

ckpt_dir

On the LoadLeveler for AIX platform, the **ckpt_dir** keyword specifies the directory that contains the checkpoint file. The actual directory used to store the checkpoint files is a combination of the value of this keyword and the value of the **ckpt_subdir** keyword.

Checkpoint files can become quite large. When specifying **ckpt_dir**, make sure that there is sufficient disk space to contain the files. Guidelines can be found in "LoadLeveler support for checkpointing jobs" on page 135.

Syntax:

ckpt_dir = *pathname*

The values for **ckpt_dir** are case sensitive.

Default value: The value of the **ckpt_dir** keyword in the class stanza of the administration file

Restriction: The keyword **ckpt_dir** is not allowed in the command file for interactive POE sessions.

Example: If checkpoint files were to be stored in the **/tmp** directory the job command file would include:

```
ckpt_dir = /tmp
```

Tips:

- When the **ckpt_dir** keyword is specified in the job command file *and* the value specifies a fully qualified directory path, this value is used to identify the "base" directory location.
- When the **ckpt_dir** keyword is specified in the job command file but the value does not specify a fully qualified directory path, the name is appended to the initial working directory and this path is used to identify the "base" directory location.
- When specified in the job command file, the value of the **ckpt_dir** keyword overrides the value of the **ckpt_dir** specified in the class stanza of the LoadLeveler administration file.
- When either **ckpt_file** or **ckpt_subdir** is specified and contains the fully qualified directory path, the value **ckpt_dir** is ignored.

ckpt_execute_dir

Specifies the directory where the job step's executable will be saved for checkpointable jobs. You may specify this keyword in either the configuration file or the job command file; different file permissions are required depending on where this keyword is set. For additional information, see "Planning considerations for checkpointing jobs" on page 136.

Syntax:

```
ckpt_execute_dir = directory
```

This directory cannot be the same as the current location of the executable file, or LoadLeveler will not stage the executable. In this case, the user must have execute permission for the current executable file.

Default value: No default value is set.

ckpt_subdir

On the LoadLeveler for AIX platform, with MetaCluster HPC enabled for checkpoint/restart capability, the **ckpt_subdir** keyword specifies the directory that will hold the files created by LoadLeveler, IBM Parallel Environment (PE), and MetaCluster HPC relating to the checkpoint and restart of the job.

If *directory* starts with a forward slash (/), this name is used as the full path name to the checkpoint directory. In this case, the **ckpt_dir** keyword is ignored. Otherwise, the actual directory used to store the checkpoint files is a concatenation of the directory specified by **ckpt_dir** and *directory*.

Syntax:

ckpt_subdir = *directory*

Note: The value for the **ckpt_subdir** keyword is case sensitive.

Default value: The default value is [*jobname.*]*job_step_id.ckpt*.

Restriction: The keyword **ckpt_subdir** is not allowed in the command file for interactive POE sessions.

Example 1: If you are storing checkpoint files in a subdirectory named **myckptdir** in the directory named by the **ckpt_dir** keyword in the class stanza of the administration file, the job command file would contain:

```
ckpt_subdir = myckptdir
```

Example 2: Alternatively, if you are naming the checkpoint subdirectory **myckptdir** and storing them in **/tmp**, the keyword in the job command file can contain:

```
ckpt_subdir = /tmp/myckptdir
```

Or, the combination of **ckpt_dir** and **ckpt_subdir** keywords can be used, producing the same result:

```
ckpt_dir = /tmp
ckpt_subdir = myckptdir
```

ckpt_time_limit

Specifies the hard or soft limit, or both limits for the elapsed time checkpointing a job can take. When the soft limit is exceeded, LoadLeveler will attempt to stop the checkpoint and allow the job to continue. If the checkpoint is not able to be stopped and the hard limit is exceeded, LoadLeveler will terminate the job.

Syntax:

ckpt_time_limit = *hardlimit*,*softlimit*

Default value: The value of the **ckpt_time_limit** keyword in the class stanza of the administration file

Examples:

```
ckpt_time_limit =  00:10:00,00:05:00
ckpt_time_limit = 12:30,7:10
ckpt_time_limit = rlim_infinity
ckpt_time_limit = unlimited
```

For additional information about limit keywords, see the following topics:
- "Syntax for limit keywords" on page 295
- "Using limit keywords" on page 94

class

Specifies the name of a job class defined locally in your cluster. You can use the **llclass** command to find out information on job classes.

Syntax:

class = *name*

Default value: If you do not specify a value for this keyword, the default job class, **No_Class**, is assigned.

Example: If you are allowed to submit jobs belonging to a class called "largejobs", your class keyword would look like the following:

```
class = largejobs
```

cluster_input_file

Specifies an individual file to be copied from the local path name to the remote path name when the job is run.

Syntax:

cluster_input_file = *local_pathname*, *remote_pathname*

where:

local_pathname
Specifies the full path name of the file to be copied from the local cluster. This file must be accessible by the submitting user on the node where the local gateway Schedd runs. *local_pathname* must be specified.

remote_pathname
Specifies the full path name the file will be copied to on the assigned cluster. This file must be accessible by the mapped user on the Schedd node of the selected cluster. *remote_pathname* must be specified. Normally the file specified by *remote_pathname* will be deleted following the job termination. It will not be deleted if the cluster selected to run the job is the same cluster where the job was submitted and *remote_pathname* resolves to the same path name specified as *local_pathname*.

If LoadLeveler fails to copy an input file to the selected cluster, the assignment of the job to the selected cluster will fail. If the cluster was assigned by the administrator using the **llmovejob** command, an error message will be displayed in the command response describing the reason for failure and the job will remain in the cluster it was in and be placed in system hold. If the cluster was assigned during job submission, the job submission fails and an error message will be displayed in the command response describing the reason for failure.

Default value: No default value is set.

cluster_list

Allows you to specify that a job is to run on a particular cluster or that LoadLeveler is to decide which cluster is best from the list of clusters specified. If this keyword is specified, it must be in the first job step of a multistep job. Any definitions in other steps are ignored.

Syntax:

cluster_list = *cluster_list*

where *cluster_list* is a blank-delimited list of cluster names or the reserved word **any**. Depending on the specified value, **cluster_list** can have one of three effects:

- Specifying a single cluster name indicates that a job is to be submitted to that cluster.
- Specifying a list of multiple cluster names indicates that the job is to be submitted to one of the clusters specified with the installation exit **CLUSTER_METRIC** choosing from the list.
- Specifying the reserved word **any** indicates the job is to be submitted to any cluster defined by the installation exit **CLUSTER_METRIC**.

Note: If a cluster list is specified using either the **llsubmit -X** command or the **ll_cluster** API, then that cluster list takes precedence over a **cluster_list** specified in the job command file.

cluster_output_file

Specifies an individual output file to be copied to the submitting cluster from the cluster selected to run the job after the job completes.

Syntax:

cluster_output_file = *local_pathname*, *remote_pathname*

where:

local_pathname
: Specifies the full path name the file will be copied to on the local cluster. This file must be accessible by the submitting user on the node where the local gateway Schedd runs. *local_pathname* must be specified.

remote_pathname
: Specifies the full path name of the file that will be copied from the assigned cluster. This file must be accessible by the mapped user on the Schedd node of the selected cluster. *remote_pathname* must be specified. Normally the file specified by *remote_pathname* will be deleted following the job termination. It will not be deleted if the cluster selected to run the job is the same cluster where the job was submitted and *remote_pathname* resolves to the same path name specified as *local_pathname*.

If LoadLeveler fails to copy an output file from a selected cluster to the local cluster during job termination, the job termination will proceed and the remote file will not be deleted. Mail will be sent to the user describing the reason for the failed copy.

Default value: No default value is set.

collective_groups

Requests the Collective Acceleration Unit (CAU) groups for the specified protocol instances of the job.

Syntax:

collective_groups = *number*

The value of the collective groups must be greater than or equal to zero. The value specified for the **collective_groups** keyword in the job command file overwrites any value specified for the **collective_groups** keyword in the administration file. If the job is not sharing the nodes with other jobs, then the

protocol instance of the job will be allocated at least *number* CAU groups. Additional CAUs can be allocated to the job step if additional CAU groups are available on the node and the node is not shared with other jobs. If the job is sharing the node with other jobs, then exactly *number* CAU groups are allocated to each protocol instance of the job.

Default value: The default value varies depending on whether the job shares the nodes with other jobs or not. If the job is not sharing the nodes with other jobs, then all the CAU groups are allocated to all the protocol instances of the job proportionally. If the job is sharing the nodes with other jobs, then zero CAU groups are allocated to the protocol instances of the job.

comment

Specifies text describing characteristics or distinguishing features of the job.

core_limit

Specifies the hard limit, soft limit, or both limits for the size of a core file. This limit is a per process limit.

Syntax:

core_limit = *hardlimit*,*softlimit*

Default value: No default value is set.

Examples:

```
core_limit = 125621,10kb
core_limit = 5621kb,5000kb
core_limit = 2mb,1.5mb
core_limit = 2.5mw
core_limit = unlimited
core_limit = rlim_infinity
core_limit = copy
```

For additional information about limit keywords, see the following topics:
- "Syntax for limit keywords" on page 295
- "Using limit keywords" on page 94

coschedule

Specifies the steps within a job that are to be scheduled and dispatched at the same time.

This keyword is supported only by the BACKFILL scheduler.

Syntax:

coschedule = yes | **no**

where **yes** implies that the step is to be coscheduled with all other steps in the job that have the value of this keyword set to **yes**. This keyword is not inherited by other job steps.

Default value: The default value is set to **no**.

Examples:

```
coschedule = yes
```

cpu_limit

Specifies the hard limit, soft limit, or both limits for the amount of CPU time that a submitted job step can use. This limit is a per process limit.

Syntax:

cpu_limit = *hardlimit*,*softlimit*

Default value: No default value is set.

Examples:

```
cpu_limit = 12:56:21,12:50:00
cpu_limit = 56:21.5
cpu_limit = 1:03,21
cpu_limit = unlimited
cpu_limit = rlim_infinity
cpu_limit = copy
```

For additional information about limit keywords, see the following topics:
- "Syntax for limit keywords" on page 295
- "Using limit keywords" on page 94

cpus_per_core

Customizes the affinity scheduling request specific to the SMT environment. This keyword specifies the number of logical CPUs per processor core that needs to be allocated to each task of a job with the processor-core affinity requirement. By default, LoadLeveler will try to allocate all of the CPUs from a core while trying to meet the processor-core affinity requirement of a job specified by the **task_affinity** keyword. This keyword can only be used along with the **task_affinity** keyword.

Syntax:

cpus_per_core = *number*

Default value: No default value is set.

data_limit

Specifies the hard limit, soft limit, or both limits for the size of the data segment to be used by the job step. This limit is a per process limit.

Syntax:

data_limit = *hardlimit*,*softlimit*

Default value: No default value is set.

Examples:

```
data_limit = ,125621
data_limit = 5621kb
data_limit = 2mb
data_limit = 2.5mw,2mb
```

For additional information about limit keywords, see the following topics:
- "Syntax for limit keywords" on page 295
- "Using limit keywords" on page 94

dependency

Specifies the dependencies between job steps. A job dependency, if used in a given job step, must be explicitly specified for that step.

Syntax:

dependency = *step_name operator value*

where:

step_name
Is the name of a previously defined job step (as described in the step_name keyword).

operator
Is one of the following:

== Equal to
!= Not equal to
<= Less than or equal to
>= Greater than or equal to
< Less than
> Greater than
&& And
|| Or

value
Is usually a number that specifies the job return code to which the *step_name* is set. It can also be one of the following LoadLeveler defined job step return codes:

CC_NOTRUN
The return code set by LoadLeveler for a job step which is not run because the dependency is not met. The value of CC_NOTRUN is 1002.

CC_REMOVED
The return code set by LoadLeveler for a job step which is removed from the system (because, for example, **llcancel** was issued against the job step). The value of CC_REMOVED is 1001.

Default value: No default value is set.

Examples: The following are examples of dependency statements:

- **Example 1:** In the following example, the step that contains this dependency statement will run if the return code from step 1 is zero:

```
dependency = (step1 == 0)
```

- **Example 2:** In the following example, step1 will run with the executable called **myprogram1**. Step2 will run only if LoadLeveler removes step1 from the system. If step2 does run, the executable called **myprogram2** gets run.

```
# Beginning of step1
# @ step_name = step1
# @ executable = myprogram1
# @ ...
# @ queue
# Beginning of step2
# @ step_name = step2
# @ dependency = step1 == CC_REMOVED
# @ executable = myprogram2
# @ ...
# @ queue
```

- **Example 3:** In the following example, step1 will run with the executable called **myprogram1**. Step2 will run if the return code of step1 equals zero. If the return code of step1 does not equal zero, step2 does not get executed. If step2 is not run, the dependency statement in step3 gets evaluated and it is determined that step2 did not run. Therefore, **myprogram3** gets executed.

```
# Beginning of step1
# @ step_name = step1
# @ executable = myprogram1
# @ ...
# @ queue
# Beginning of step2
# @ step_name = step2
# @ dependency = step1 == 0
# @ executable = myprogram2
# @ ...
# @ queue
# Beginning of step3
# @ step_name = step3
```

```
# @ dependency = step2 == CC_NOTRUN
# @ executable = myprogram3
# @ ...
# @ queue
```

- **Example 4:** In the following example, the step that contains step2 returns a non-negative value if successful. This step should take into account the fact that LoadLeveler uses a value of 1001 for CC_REMOVED and 1002 for CC_NOTRUN. This is done with the following dependency statement:

  ```
  dependency = (step2 >= 0) && (step2 < CC_REMOVED)
  ```

dstg_environment

Specifies the environment that must be passed to the data staging scripts. **dstg_environment** is similar to the **environment** keyword and the usage rules for this keyword are identical to the **environment** keyword. For example, with both keywords, **COPY_ALL** will copy all the environment variables from your user environment.

Syntax:

dstg_environment = *env1; env2* **...;COPY_ALL**

Default value: No default value is set.

dstg_in_script

Specifies the script or executable that is used to stage in data. You are only allowed to have one instance of this keyword in a job command file. Any arguments must be included as part of the value specified for this keyword. No arguments can be separately or explicitly passed to the script or executable.

Note: If you specify the **dstg_in_wall_clock_limit** keyword, then you must specify a value for the **dstg_in_script** keyword. If you do not do so, the job will not be submitted to LoadLeveler.

Syntax:

dstg_in_script = *name*

Default value: No default value is set.

dstg_in_wall_clock_limit

Specifies both the hard wall clock limit and the soft wall clock limit for an inbound data staging script. You can assign the value in either seconds or using the *HH:MM:SS,HH:MM:SS* format. The usage rules will be the same as those for the **wall_clock_limit** keyword.

Note: If you specify the **dstg_in_wall_clock_limit** keyword, then you must specify a value for the **dstg_in_script** keyword. If you do not do so, the job will not be submitted to LoadLeveler.

Syntax:

dstg_in_wall_clock_limit = *HH:MM:SS*,*HH:MM:SS*

Default value: No default value is set.

For additional information about limit keywords, see the following topics:
- "Syntax for limit keywords" on page 295
- "Using limit keywords" on page 94

dstg_node

Specifies which node to use for running the staging steps.

Syntax:

dstg_node = **any** **| master| all**

Default value: any

The possible values for this keyword are:

any Specifies a shared staging area is accessible from any executing machine. The staging task will be executed on any node in the cluster that has an available data staging initiator and any other required data staging resources.

master
Specifies the data staging job step must run on the master node of the application job step. If this value is specified, the central manager will schedule the application job step first and get the host list on which the step will run. It will then ensure that data staging resources are available on the master node and schedule the data staging job step to run on the master node. This option can be specified only if the **dstg_time** configuration keyword is set to **JUST_IN_TIME**.

all Specifies that a data staging task needs to run on each node allocated to the application job step. A possible use of this value is when data has to be staged to or from the local disk on the executing machine. The central manager will ensure that resources are available on the required nodes for data staging. The central manager will also schedule the application job step in the future to determine the nodes that have to be used and then schedule the data staging master task. Usually, the data staging master task is a POE command or an MPI script that spawns data staging tasks on all the selected nodes. This option can be specified only if the **dstg_time** configuration keyword is set to **JUST_IN_TIME**.

Note: If you are using the **master** or **all** options for this keyword, the administrators should setup one or more data staging initiators on every compute node in the cluster. Although this is not mandatory, if you do not set up the initiators, jobs may remain in idle states for extended periods.

dstg_out_script
Specifies the script or executable that is used to stage out data. You are only allowed to have one instance of this keyword in a job command file. Any arguments must be included as part of the value specified for this keyword. No arguments can be separately or explicitly passed to the script or executable.

Note: If you specify the **dstg_out_wall_clock_limit** keyword, then you must specify a value for the **dstg_out_script** keyword. If you do not do so, the job will not be submitted to LoadLeveler.

Syntax:

```
dstg_out_script = name
```

Default value: No default value is set.

dstg_out_wall_clock_limit
Specifies the hard wall clock limit and the soft wall clock limit for an outbound data staging script. You can assign a value using seconds or the *HH:MM:SS,HH:MM:SS* format. The usage rules for **dstg_out_wall_clock_limit** are the same as those for **wall_clock_limit** keyword.

Note: If you specify the **dstg_out_wall_clock_limit** keyword, then you must specify a value for the **dstg_out_script** keyword. If you do not do so, the job will not be submitted to LoadLeveler.

Syntax:

dstg_out_wall_clock_limit = *HH:MM:SS*,*HH:MM:SS*

Default value: No default value is set.

For additional information about limit keywords, see the following topics:
- "Syntax for limit keywords" on page 295
- "Using limit keywords" on page 94

dstg_resources

Specifies the quantities of consumable resources consumed by each task of both the inbound and outbound data staging job steps. These resources may be either machine resources or floating resources. The syntax and usage of this keyword are the same as those used for the **resources** keyword in the job command file. However, **dstg_resources** applies to data staging job steps while the **resources** keyword applies to all other job steps. The **dstg_resources** are checked at job submit time to ensure their availability. They are not enforced at runtime.

Note: The limits specified in the **max_resources** keyword of the **data_stage** class stanza also apply to **dstg_resources**. If the resources you requested for the data staging step of the job exceed what is allowed in the **max_resources** keyword, the job will not be submitted and an error message stating this will be displayed by the **llsubmit** command.

Syntax:

dstg_resources = name(*count*) *name*(*count*) **...** *name*(*count*)

Default value: If you do not specify **dstg_resources** in the job command file, then the **default_resources** applicable to the **data_stage** class in the administration file will be applied to each task of a data staging job step.

energy_policy_tag

Specifies the energy policy tag name for the job. LoadLeveler uses this keyword to identify the energy data for a job. Be sure to use a unique tag name for the job when you submit it for the first time. On subsequent submissions the energy data identified by the tag name is used to determine the CPU frequency the job will run with.

Syntax:

energy_policy_tag = *tag_name*

where:

tag_name
Is a unique energy policy name that you set in the job command file. If the tag name does not contain a user name as a prefix (delimited by a period), then LoadLeveler will add your user name as a prefix in the specified tag name.

Default value: No tag name is set.

Examples:

1. To set the **energy_policy_tag** in job command file, enter:

```
# tom.cmd
# @ energy_policy_tag = long_running_job
```

   In this example, the energy policy tag is **tom.long_running_job**.
2. For user Tom to use the energy policy tag that was generated by John, enter:

```
# tom.cmd
# @ energy_policy_tag = john.IO_job
```

In this example, because Tom wants to use the energy tag that was generated by John, John has to have previously generated the **IO_job** energy tag by submitting a job specifying it, otherwise this job is not successfully submitted by LoadLeveler when Tom wants to use John's energy tag and the tag does not exist in the database.

energy_saving_req

Specifies the minimum energy savings percentage that is required. The value can be an integer between 0 and 99, inclusive.

Syntax:

energy_saving_req = *percent_number*

where:

percent_number
Is the required energy savings percentage. This keyword cannot be used with the **max_perf_decrease_allowed** keyword. The job is not successfully submitted if the required energy savings percentage cannot be satisfied.

Default value: 0

Examples:

In this example, the energy tag content for userA is shown:

```
  userA.logn_running_job:
      Generated by: c111bc4n13.ppd.pok.ibm.com.91.0
    Last used Time: Tue May 10 05:00:01 EDT 2011
              User: userA
 Nominal Frequency: 3.3 GHZ
Energy Consumption: 0.8 Kwh
    Execution Time: 950 Seconds
Frequency(GHZ) EstedEnergyCons(Kwh) EngSaving(%) EstedTime(Seconds) PerfDeg(%)
3.1                    0.75           6            1000              5
2.8                    0.65           19           1050              11
2.6                    0.6            25           1100              16
2.2                    0.5            38           1150              21
```

UserA required saving at least 20% energy consumption as specified in the job command file:

```
# userA.cmd
# @ energy_policy_tag = long_running_job
# @ energy_saving_req = 20
```

The submitted job will be run in the frequency 2.6 GHZ.

UserA required saving at least 40% energy consumption as specified in the job command file:

```
# userA.cmd
# @ energy_policy_tag = long_running_job
# @ energy_saving_req = 40
```

In this instance, the submitted job is rejected because there is no suitable frequency in the energy tag that can meet the energy saving requirement.

env_copy

Specifies whether environment variables for a batch or interactive parallel job are copied to all executing nodes, or to only the master node. When **all** is specified either explicitly or by default, any environment variables (specified

by the **environment** keyword in the job command file) will be copied to all nodes where the job step runs. When **master** is specified, the environment variables will be copied only to the node selected to run the master task of the parallel job.

Although a LoadLeveler administrator may set this keyword in one or more class, group, or user stanzas in the administration file, an explicit setting in the job command file overrides any settings in the administration file that are relevant for the parallel job.

LoadLeveler ignores this keyword if it is set for a serial job.

Syntax:

```
env_copy = all | master
```

Default value: LoadLeveler uses the default value **all** only when both of the following conditions are true:

- The **env_copy** keyword is not specified in the job command file.
- The **env_copy** keyword is not specified in any class, group, or user stanza that is relevant to the parallel job.

environment

Specifies login initial environment variables set by LoadLeveler when your job step starts. If the same environment variables are set in the user's initialization files (such as the .profile), those set by the login initialization files will supersede those set by LoadLeveler.

You may use the **env_copy** keyword to instruct LoadLeveler to copy these environment variables to all executing nodes, or to only the master executing node.

Syntax:

```
environment = env1 ; env2 ; ...
```

Separate environment specifications (*env1*, *env2*, and so on) with semicolons. An environment specification may be one of the following:

COPY_ALL
: Specifies that all the environment variables from your shell be copied.

$var
: Specifies that the environment variable *var* be copied into the environment of your job when LoadLeveler starts it.

!var
: Specifies that the environment variable *var* not be copied into the environment of your job when LoadLeveler starts it. This specification is most useful together with COPY_ALL.

var=value
: Specifies that the environment variable *var* be set to the value "value" and copied into the environment of your job when LoadLeveler starts it.

  When processing the string you specify for *var*, LoadLeveler first removes any leading or trailing blanks, and copies the remaining string, as is, into the environment.

Default value: No default value is set.

Additional considerations:

If you specify the **environment** job command file keyword with **COPY_ALL**, the **$USER** and **$HOME** environment variables from your shell are not copied and set when your job step starts. The **$USER** and **$HOME** environment variables of the user ID on the executing node will be set. If you explicitly specify **$USER** or **$HOME** it will be copied and set when your job step starts.

If more than one environment specification is defined for the same environment keyword, the rightmost specification takes precedence. For example if you specify:

```
environment = COPY_ALL; USER=jsmith
```

The **$USER** environment variable will be set to `jsmith`.

However, if you specify:

```
environment = USER=jsmith; COPY_ALL
```

The **$USER** environment variable is not set to `jsmith`. Instead, the **$USER** environment variable of the user ID on the executing node is set.

If the **executable** keyword is not specified, the job command file is run as a shell script. In this case, LoadLeveler initializes the environment as described previously and then starts the **shell** command. Any environment variable set during shell startup overrides values initialized by LoadLeveler.

Examples:

- This example illustrates how to specify that LoadLeveler is to copy all the environment variables from your shell except for `env2`:

  ```
  environment = COPY_ALL; !env2;
  ```

- This example illustrates how LoadLeveler processes the string you specify with *var*: If you specify the following:

  ```
  environment = env3 = "quoted string"; env4 = imbedded blanks;
  ```

  LoadLeveler uses these values:
  - For `env3`: **"quoted string"**
  - For `env4`: **imbedded blanks**

error

Specifies the name of the file to use as standard error (stderr) when your job step runs.

Syntax:

error = *filename*

Default value: If you do not specify a value for this keyword, the file **/dev/null** is used.

Example:

```
error = $(jobid).$(stepid).err
```

executable

Identifies the name of the program to run, which can be a shell script or a binary. For parallel jobs, **executable** must be the parallel job launcher (POE or mpirun), or the name of a program that invokes the parallel job launcher.

Note that the **executable** statement automatically sets the **$(base_executable)** variable, which is the file name of the executable without the directory component. See Example 2 in topic "Examples: Job command files" on page 173 for an example of using the **$(base_executable)** variable.

Syntax:

executable = *name*

Default value: If you do not include this keyword, then it will default to the job command file that is being submitted, and LoadLeveler will assume that the file is a valid shell script.

Examples:

- `# @ executable = a.out`
- `# @ executable = /usr/bin/poe` (for POE jobs)

file_limit

Specifies the hard limit, soft limit, or both limits for the size of a file. This limit is a per process limit.

Syntax:

file_limit = *hardlimit*,*softlimit*

Default value: No default value is set.

Example:

```
file_limit = 100pb,50tb
```

For additional information about limit keywords, see the following topics:
- "Syntax for limit keywords" on page 295
- "Using limit keywords" on page 94

first_node_tasks

Specifies a different task count for the first node assigned to a job step using **first_node_tasks**. All remaining nodes will run **tasks_per_node** tasks. The **first_node_tasks** keyword can only be specified in conjunction with **node** and **tasks_per_node**; it cannot be specified with the **total_tasks** keyword.

When **first_node_tasks** is used, a maximum node specification is not permitted. For example, **node = 6** is acceptable but **node = 6, 12** is not.

A value of 0 means that the first node will get the same number of tasks as all other nodes (**tasks_per_node** tasks). A negative value will result in an error and the job will not be submitted.

Syntax:

first_node_tasks = *number*

Default value: The default value is 0.

Example:

In the following example, a total of 7 machines are selected for the job step. The first machine will run only 1 task, task ID 0, and the remaining 6 machines will run 16 tasks each. The total number of tasks in this job step is 97.

```
#@ node = 7
#@ first_node_tasks = 1
#@ tasks_per_node = 16
```

group

Specifies the LoadLeveler group.

Syntax:

group = *group_name*

Default value: If you do not specify a value for this keyword, the default group for the user is used, If a default group is not defined for the user, LoadLeveler uses the group, **No_Group**.

Example:

```
group = my_group_name
```

hold

Specifies whether you want to place a hold on your job step when you submit it. There are three types of holds:

user Specifies user hold

system
Specifies system hold

usersys
Specifies user and system hold

To remove the hold on the job, use the **llhold -r** command.

Syntax:

hold = user | system | usersys

Default value: No default is set, which means that no hold is requested.

Example: To put a user hold on a job, the keyword statement would be:

```
hold = user
```

host_file

Specifies the name of the file containing the host list for task allocation.

Syntax:

host_file = *host_list_file_name*

If a full path name is not specified, the *host_file_name* will be under the current working directory. The *host_file_name* is an ASCII file that must contain one host name per line (using a new line separator).

Host file usage notes:

- Leading and trailing tabs and spaces will be removed
- Blank lines are deleted
- Comment lines that start with a # will be skipped
- Tabs or spaces before the first comment line are allowed
- No more than one word per line is allowed; otherwise, the entire line will be treated as a host name

Default value: The default is empty (a NULL file pointer) meaning no host file input.

Example: To specify host file **my_host_file** in the current working directory in the job command file, the keyword statement would be:

```
host_file = my_host_file
```

image_size

Specifies the maximum virtual image size to which your program will grow during execution. LoadLeveler tries to execute your job steps on a machine that has enough resources to support executing and checkpointing your job step. If your job command file has multiple job steps, the job steps will not necessarily run on the same machine, unless you explicitly request that they do.

If you underestimate the image size of your job step, your job step may crash due to the inability to acquire more address space. If you overestimate the image size, LoadLeveler may have difficulty finding machines that have the required resources.

Syntax:

image_size = *number*

where *number* must be a positive integer. If you do not specify the units associated with this keyword, LoadLeveler uses the default unit, which is kilobytes. For a list of allowable units, see the resources keyword description.

Default value: If you do not specify the image size of your job command file, the image size is that of the executable.

Example: To set an image size of 11 KB, the keyword statement would be:

```
image_size = 11
```

For additional information about limit keywords, see the following topics:
- "Syntax for limit keywords" on page 295
- "Using limit keywords" on page 94

imm_send_buffers

Requests a number of immediate send buffers for each window allocated for each protocol instance of the job.

Syntax:

```
imm_send_buffers=number
```

The value of the immediate send buffers must be greater than or equal to zero. The value of the immediate send buffers is inherited from all the protocol instances of the job step unless the individual protocol instances are specified with their own immediate send buffers. If the job is sharing nodes with other jobs, then exactly *number* immediate send buffers are allocated to each window assigned to each protocol instance of the job. If the job is not sharing the nodes with other jobs then at least *number* immediate send buffers are allocated to each window assigned to each protocol instance of the job. Additional immediate send buffers can be distributed evenly to the windows assigned to the job if the nodes are not shared with other jobs.

Default value: The default value varies depending on whether the job shares the nodes with other jobs or not. If the job is not sharing the nodes with other jobs, then all the immediate send buffers are allocated to all the protocol instances of the job proportionally. If the job is sharing the nodes with other jobs, then one immediate send buffer is assigned for each window assigned to the job.

initialdir

Specifies the path name of the directory to use as the initial working directory during execution of the job step. File names mentioned in the command file which do not begin with a slash **(/)** are relative to the initial directory. The initial directory must exist on the submitting machine as well as on the machine where the job runs.

Syntax:

```
initialdir = pathname
```

Note: When operating in a multicluster environment, access to **initialdir** will be verified on the cluster selected to run the job. If access to **initialdir** fails, the submission or move job will fail.

Default value: If you do not specify a value for this keyword, the initial directory is the current working directory at the time you submitted the job.

Example:

```
initialdir  = /var/home/mike/ll_work
```

input

Specifies the name of the file to use as standard input (stdin) when your job step runs.

Syntax:

input = *filename*

Default value: If you do not specify an input file, LoadLeveler uses the file **/dev/null**

Example:

```
input = input.$(process)
```

island_count

Specifies the minimum and maximum number of islands to select for this job step, as well as whether the minimum or maximum is preferred.

Syntax:

island_count = *number*[**,***number*]

Where the two number values are the preferred and acceptable number of islands to select for this job step. The first value is the preferred allocation. A value of -1 is used to represent all islands in the cluster. Other than this special value of -1, both numbers must be greater than zero. This keyword is only used when **node_topology = island**, otherwise the **island_count** keyword is ignored. See the **node_topology** keyword for more information.

Default value: The default value is 1,1, which specifies that all machines allocated to the job step must come from a single island.

Example 1:

```
island_count = -1,4
```

In example 1, the scheduler will try to schedule this job step to run across all islands in the cluster. If there are not machines available in every island, then fewer islands may be allocated, down to a minimum of 4 islands.

Example 2:

```
island_count = 1,2
```

In example 2, the scheduler will try to schedule this job step to run within a single island. If there are not enough available machines within 1 island to run the job step, then machines from more than 1 island may be used, up to a maximum of 2.

Example 3:

```
island_count = 4
```

Example 3 shows that this is the same as specifying `island_count=4,4,` and it means that the scheduler will use exactly 4 islands to run the job step. If there are less than 4 islands with available machines, then the job step will remain Idle. If there are more than 4 islands with available machines, only 4 islands will be used to run the job step.

job_cpu_limit

Specifies the hard limit, soft limit, or both limits for the CPU time used by all processes of a serial job step. For example, if a job step runs as multiple processes, the total CPU time consumed by all processes is added and controlled by this limit.

For parallel job steps, LoadLeveler enforces these limits differently. Parallel job steps usually have tasks running on several different nodes and each task can have several processes associated with it. In addition, the parallel tasks running on a node are descendants of a **LoadL_starter** process. Therefore, if you specify a hard or soft CPU time limit of S seconds and if a **LoadL_starter** has N tasks running under it, then all tasks associated with that **LoadL_starter** will be terminated if the total CPU time of the **LoadL_starter** process and its children is greater than S*N seconds.

If several **LoadL_starter** processes are involved in running a parallel job step, then LoadLeveler enforces the limits associated with the job_cpu_limit keyword independently for each **LoadL_starter**. LoadLeveler determines how often to check the job_cpu_limit by looking at the values for JOB_LIMIT_POLICY and JOB_ACCT_Q_POLICY. The smaller value associated with these two configuration keywords sets the interval for checking the job_cpu_limit. For more information on JOB_LIMIT_POLICY and JOB_ACCT_Q_POLICY see "Collecting job resource data on serial and parallel jobs" on page 66.

Syntax:

job_cpu_limit = *hardlimit*,*softlimit*

Default value: No default is set.

Example:

```
job_cpu_limit = 12:56,12:50
```

For additional information about limit keywords, see the following topics:
- "Syntax for limit keywords" on page 295
- "Using limit keywords" on page 94

job_name

Specifies the name of the job. This keyword must be specified in the first job step. If it is specified in other job steps in the job command file, it is ignored.

The job_name only appears in the long reports of the **llq**, **llstatus**, and **llsummary** commands, and in mail related to the job.

Syntax:

job_name = *job_name*

You can name the job using any combination of letters, numbers, or both.

Default value: No default value is set.

Example:

```
job_name = my_first_job
```

job_type

Specifies the type of job step to process.

Syntax:

```
job_type = serial | parallel | bluegene | MPICH
```

This keyword is inherited by each job step in the job command file.

Default value: serial

large_page

Specifies whether or not a job step requires Large Page support from AIX.

Restriction: Large Page memory is not supported in LoadLeveler for Linux. In this case, specifying **M** would cause the job to never be sent.

Syntax:

```
large_page = value
```

where *value* can be **Y**, **M**, or **N**. **Y** informs LoadLeveler to use Large Page memory, if available, but to otherwise use regular memory. **M** means use of Large Page memory is mandatory.

Default value: N, which means to not use Large Page memory.

Example: To ask LoadLeveler to use Large Page memory for the job step, if available, specify:

```
large_page = Y
```

ll_res_id

Specifies the reservation to which the job is submitted.

This keyword, if specified, overrides the LL_RES_ID environment variable.

If the keyword is present in the job command file with no value, the job will not be bound to any reservation. If the keyword is not present in the job command file, the LL_RES_ID environment variable determines the reservation to which the job is submitted.

The value of this keyword is ignored when the job command file is used by the **llmkres -f** or the **llchres -f** command. The reservation ID can be obtained from the **llqres** command or when the **llmkres** command is issued.

The format of the reservation identifier is [*host*.]*rid*[.**r**[.*oid*]].

where:

- *host* is the name of the machine that assigned the reservation identifier.
- *rid* is the number assigned to the reservation when it was created. An *rid* is required.
- **r** indicates that this is a reservation ID (**r** is optional if *oid* is not specified).
- *oid* is the occurrence ID of a recurring reservation (*oid* is optional).

When `oid` is specified, the job step will not be considered for scheduling until that occurrence of the reservation becomes active. The step will remain in Idle state during all earlier occurrences. When `oid` is not specified, the job will be bound to the first occurrence of the reservation, including a currently active occurrence.

Syntax:

```
ll_res_id = reservation ID
```

Default value: No default value is set.

locks_limit

Specifies the hard limit, soft limit, or both limits for the number of file locks that the submitted job can use. This limit is a per process limit.

Syntax:

```
locks_limit = hardlimit,softlimit
```

Default value: No default value is set.

Example:

```
locks_limit = 200,65
```

For additional information about limit keywords, see the following topics:
- "Syntax for limit keywords" on page 295
- "Using limit keywords" on page 94

max_perf_decrease_allowed

Specifies the performance degradation percentage that is acceptable for the job. The value that can be specified is an integer between 0 and 99, inclusive.

Syntax:

max_perf_decrease_allowed = *percent_number*

where:

percent_number
Is the performance degradation percentage. This keyword cannot be used at the same time as the **energy_saving_req** keyword.

Default value: 0

Note: This keyword is ignored if the **energy_policy_tag** is not valid.

Examples:

In this example, userA set the performance degradation to 20% in the job command file:

```
# userA.cmd
# @ energy_policy_tag = long_running_job
# @ max_perf_decrease_allowed = 20
```

The content for the energy tag **long_running_job** is:

```
  long_running_job:
      Generated by: c111bc4n13.ppd.pok.ibm.com.91.0
    Last used Time: Tue May 10 05:00:01 EDT 2011
              User: userA
 Nominal Frequency: 3.3 GHZ
Energy Consumption: 0.8 Kwh
    Execution Time: 950 Seconds
Frequency(GHZ) EstedEnergyCons(Kwh) EngSaving(%) EstedTime(Seconds) PerfDeg(%)
  3.1                   0.75           6            1000              5
  2.8                   0.65           19           1050              11
  2.6                   0.6            25           1100              16
  2.2                   0.5            38           1150              21
```

In this example, the maximum performance degradation of the tag is 21% and the maximum performance degradation allowed by userA is 20% in the job command file. The job will run at the frequency 2.6 GHZ.

mcm_affinity_options

Specifies the affinity options for a job.

Syntax:

mcm_affinity_options = *affinity_option*

where *affinity_option* is a blank-delimited list of one, two, or three keywords chosen from the three groupings of keywords in the list that follows. Only one option from each group may be specified.

task affinity options
The following options are task affinity options. These options are mutually exclusive.

mcm_accumulate
Specifying this option tells the central manager to accumulate tasks on the same MCM whenever possible.

mcm_distribute
Specifying this option tells the central manager to distribute tasks across all available MCMs on a machine.

memory affinity options
The following options are memory affinity options. These options are mutually exclusive.

mcm_mem_none
When you specify **mcm_mem_none**, the job does not request memory affinity.

mcm_mem_pref
When you specify **mcm_mem_pref**, the job requests memory affinity as a preference.

mcm_mem_req
When you specify **mcm_mem_req**, the job requests memory affinity as a requirement.

adapter affinity options
The following options are adapter affinity options. These options are mutually exclusive.

mcm_sni_none
Specifying this option indicates the job has no adapter affinity requirement.

mcm_sni_pref
Specifying this option indicates the job requests adapter affinity.

mcm_sni_req
Specifying this option indicates the job requires adapter affinity. When the **mcm_sni_req** option is specified, the network usage should be requested as **not_shared**.

Your job containing the keyword **mcm_affinity_options** will not be submitted to LoadLeveler unless the **rset** keyword is set to **RSET_MCM_AFFINITY**.

Default value: Table 57 describes the default values for **mcm_affinity_options** depending on the type of affinity (MCM affinity or task affinity) and network requirements.

Table 57. mcm_affinity_options default values

If the following keywords are specified in the job command file:	Memory Affinity	Task Allocation	Adapter Affinity
#@ rset = rset_mcm_affinity #@ network.*protocol* = sn_all,,,,	mcm_mem_req	mcm_accumulate	mcm_sni_none
#@ rset = rset_mcm_affinity #@ network.*protocol* = sn_single,,,,	mcm_mem_req	mcm_accumulate	mcm_sni_pref
#@ task_affinity = **cpu**(*number*) | **core**(*number*) #@ network.*protocol* = sn_all,,,,	mcm_mem_pref	mcm_accumulate	mcm_sni_none
#@ task_affinity = **cpu**(*number*) | **core**(*number*) #@ network.*protocol* = sn_single,,,,	mcm_mem_pref	mcm_accumulate	mcm_sni_pref

Examples:

1. This example shows how to have a job set memory affinity as a requirement, adapter affinity as a preference, and MCM task allocation method as distribute.

```
mcm_affinity_options = mcm_mem_req mcm_sni_pref mcm_distribute
```

2. This example shows how to have a job set adapter affinity as a requirement:

```
mcm_affinity_options = mcm_sni_req
...
network.mpi = sn_single, not_shared, us,, instances=1
...
```

memlock_limit

Specifies the hard limit, soft limit, or both limits for the memory that can be locked by each process of the submitted job. This limit is a per process limit.

Syntax:

memlock_limit = *hardlimit*,*softlimit*

Default value: No default value is set.

Example:

```
memlock_limit = 1gb,256mb
```

For additional information about limit keywords, see the following topics:
- "Syntax for limit keywords" on page 295
- "Using limit keywords" on page 94

network

Specifies communication protocols, adapters, and their characteristics. You need to specify this keyword when you want a task of a parallel job step to request a specific adapter that is defined in the LoadLeveler administration file. You do not need to specify this keyword when you want a task to access a shared, default adapter through TCP/IP. (A default adapter is an adapter whose name matches a machine stanza name.)

Note that you cannot specify both the **network** statement and the **Adapter** requirement in a job command file. Also, the value of the **network** keyword applies only to the job step in which you specify the keyword. (That is, this keyword is not inherited by other job steps.)

This keyword is supported by the BACKFILL and API schedulers.

Syntax:

network.*protocol*[(*number*)]=*type*[,*usage*[,*mode*[,*comm_level*[,**instances**=<*number*|**max**> \
[,**rcxtblocks**=*number*[,**collective_groups**=*number*[,**imm_send_buffers**=*number*] \
[,**endpoints**=*number*]]]]]]]

where:

protocol
: Specifies any communication protocols that are supported by Parallel Environment Runtime Edition.

number
: The protocol can include an optional number of contexts, expressed as:

  protocol(*number*)

  For more information, see:
  - *IBM Parallel Environment Runtime Edition for AIX: Operation and Use*
  - *IBM Parallel Environment Runtime Edition for AIX: MPI Programming Guide*

type This field is required and specifies one of the following:

sn_single
When used for switch adapters, it specifies that LoadLeveler use a common, single switch network.

sn_all Specifies that striped communication should be used over all available switch networks. The networks specified must be accessible by all machines selected to run the job. For more information on striping, see "Submitting jobs that use striping" on page 188.

The following are optional and if omitted their position must be specified with a comma:

usage Specifies whether the adapter can be shared with tasks of other job steps. Possible values are **shared**, which is the default, or **not_shared**. If **not_shared** is specified, LoadLeveler can only guarantee that the adapter will not be shared by other jobs running on the same OSI. If the adapter is shared by more than one OSI, LoadLeveler cannot guarantee that the adapter is not shared with jobs running on a different OSI.

mode Specifies the communication subsystem mode used by the communication protocol that you specify, and can be either **IP** (Internet Protocol), which is the default, or **US** (User Space). Note that each instance of the US mode requested by a task running on switch adapters requires an adapter window. For example, if a task requests both the MPI and LAPI protocols such that both protocol instances require US mode, two adapter windows will be used.

comm_level

Note: This keyword is obsolete and will be ignored, however it is being retained for compatibility and because the parameters in the network statement are positional.

The **comm_level** keyword should be used to suggest the amount of inter-task communication that users *expect* to occur in their parallel jobs. This suggestion is used to allocate adapter device resources. Specifying a level that is higher than what the job actually needs will not speed up communication, but may make it harder to schedule a job (because it requires more resources). The **comm_level** keyword can only be specified with **US** mode. The three communication levels are:

LOW Implies that minimal inter-task communication will occur.

AVERAGE
This is the default value. Unless you know the specific communication characteristics of your job, the best way to determine the comm_level is through trial-and-error.

HIGH Implies that a great deal of inter-task communication will occur.

instances=<*number* **| max>**
If **instances** is specified as a number, it indicates the number of parallel communication paths made available to the protocol on each network. The number actually used will depend on the implementation of the protocol subsystem. If **instances** is specified by **max**, the actual value used is determined by the **MAX_PROTOCOL_INSTANCES** for the class to which the job is submitted. The default value for **instances** is 1.

For the best performance set **MAX_PROTOCOL_INSTANCES** so that the communication subsystem uses every available adapter before it reuses any of the adapters.

rcxtblocks=*number*
Integer value specifying the number of user rCxt blocks requested for each window used by the associated protocol. The values of this keyword are not inherited between steps in a multistep job.

Note: Use of this keyword will prevent adapters from the SP Switch2 family from being used by the job.

collective_groups=*number*
Requests the Collective Acceleration Unit (CAU) groups for the specified protocol instances of the job.

The value of the collective groups must be greater than or equal to zero. The value specified for the **collective_groups** keyword in the job command file overwrites any value specified for the **collective_groups** keyword in the administration file. If the job is not sharing the nodes with other jobs, then the protocol instance of the job will be allocated at least *number* CAU groups. Additional CAUs can be allocated to the job step if additional CAU groups are available on the node and the node is not shared with other jobs. If the job is sharing the node with other jobs, then exactly *number* CAU groups are allocated to each protocol instance of the job.

The default value varies depending on whether the job shares the nodes with other jobs or not. If the job is not sharing the nodes with other jobs, then all the CAU groups are allocated to all the protocol instances of the job proportionally. If the job is sharing the nodes with other jobs, then zero CAU groups are allocated to the protocol instances of the job.

imm_send_buffers=*number*
Requests a number of immediate send buffers for each window allocated for each protocol instance of the job.

The value of the immediate send buffers must be greater than or equal to zero. The value of the immediate send buffers is inherited from all the protocol instances of the job step unless the individual protocol instances are specified with their own immediate send buffers. If the job is sharing nodes with other jobs, then exactly *number* immediate send buffers are allocated to each window assigned to each protocol instance of the job. If the job is not sharing the nodes with other jobs then at least *number* immediate send buffers are allocated to each window assigned to each protocol instance of the job. Additional immediate send buffers can be distributed evenly to the windows assigned to the job if the nodes are not shared with other jobs.

endpoints= 1 | 2 | 4 | 8 | 16 | 32 | 64 | 128
Requests the number of endpoints that can be used by each task per protocol instance.

The *number* value must be a power of 2 and no greater than 128 (that is, from {1, 2, 4, 8, 16, 32, 64, 128}). If the value specified is not a power of 2, the next higher power of 2 is used and a warning message is issued. The value of **endpoints** is inherited to all the protocol instances of the job step unless the individual protocol instances are specified with their own endpoints.

Note: If multiple network statements with endpoints are used, all the endpoints values must be the same.

Default value: If there is no **network** statement in the job command file, the **default_network.MPI** and **default_network.LAPI** keywords defined in the class stanza will serve as the defaults of **network.MPI** and **network.LAPI** in the job command file. If you do not specify **default_network** in the class stanza and you do not specify the **network** keyword in the job command file, LoadLeveler allows the task to access a shared default adapter through TCP/IP. The default adapter is the adapter associated with the machine name.

Examples:

- **Example 1:** To use the MPI protocol with an adapter in User Space mode without sharing the adapter, enter the following:

  ```
  network.MPI = sn_single,not_shared,US,HIGH
  ```

- **Example 2:** To use the MPI protocol with a shared adapter in IP mode, enter the following:

  ```
  network.MPI = sn_single,,IP
  ```

  Because a shared adapter is the default, you do not need to specify **shared**.

- **Example 3:** A communication level can only be specified if User Space mode is also specified:

  ```
  network.MPI = sn_single,,US,AVERAGE
  ```

  Note that LoadLeveler can ensure that an adapter is dedicated (not shared) if you request the adapter in US mode, since any user who requests a user space adapter must do so using the **network** statement. However, if you request a dedicated adapter in IP mode, the adapter will only be dedicated if all other LoadLeveler users who request this adapter do so using the **network** statement.

node

Specifies the minimum and maximum number of nodes requested by a job step. You must specify at least one of these values. The value of the **node** keyword applies only to the job step in which you specify the keyword. (That is, this keyword is not inherited by other job steps.)

When you use the **node** keyword together with the **total_tasks** keyword, the *min* and *max* values you specify on the **node** keyword must be equal, or you must specify only one value. For example:

```
node = 6
total_tasks = 12
```

This keyword is supported by the BACKFILL and API schedulers.

Syntax:

node = [*min*][,max]

where:

min Specifies the minimum number of nodes requested by the job step.

max Specifies the maximum number of nodes requested by the job step. The maximum number of nodes a job step can request is limited by the **max_node** keyword in the administration file (provided this keyword is specified). That is, the maximum must be less than or equal to any **max_node** value specified in a user, group, or class stanza.

Default value: The default value for *min* is 1; the default value for *max* is the *min* value for this keyword.

Example: To specify a range of six to twelve nodes, enter the following:

```
node = 6,12
```

To specify a maximum of seventeen nodes, enter the following:

```
node =  ,17
```

For information about specifying the number of tasks you want to run on a node, see "Task-assignment considerations" on page 186 and the **task_geometry**, **tasks_per_node**, and **total_tasks** job command file keywords.

node_resources

Specifies quantities of the consumable resources consumed by each node of a job step. The resources must be machine resources; floating resources cannot be requested with the **node_resources** keyword.

Syntax:

```
node_resources =name(count) name(count) ... name(count)
```

where *name(count)* is one of the following:

- An administrator-defined name and count
- **ConsumableCpus***(count)*
- **ConsumableMemory***(count units)*
- **ConsumableVirtualMemory***(count units)*
- **ConsumableLargePageMemory***(count units)*

The count for each specified resource must be an integer greater than or equal to zero, except for the following instances in which the integer must be greater than zero:

- **ConsumableMemory**
- **ConsumableVirtualMemory**
- **ConsumableCpus** when the enforcement policy is *hard* or *soft*

ConsumableCpus can have a value of zero when the administrator has not requested that consumable resources be enforced, or when the enforcement policy is *shares*.

When you set **ConsumableCpus** to zero, the meaning varies depending on whether use is being enforced. With no enforcement, zero means that the job is requesting a negligible amount of CPU. With an enforcement policy of *shares*, it means that the job is requesting a small percentage of available shares.

If the count is not valid, LoadLeveler will issue a message and the job will not be submitted. The allowable units are those normally used with LoadLeveler data limits:

```
b bytes
w words      (4 bytes)
kb kilobytes (2**10 bytes)
kw kilowords (2**12 bytes)
mb megabytes (2**20 bytes)
mw megawords (2**22 bytes)
gb gigabytes (2**30 bytes)
gw gigawords (2**32 bytes)
tb terabytes (2**40 bytes)
tw terawords (2**42 bytes)
pb petabytes (2**50 bytes)
pw petawords (2**52 bytes)
eb exabytes  (2**60 bytes)
ew exawords  (2**62 bytes)
```

ConsumableMemory, **ConsumableVirtualMemory**, and **ConsumableLargePageMemory** values are stored in megabytes (MB) and are rounded up. For **ConsumableMemory** and **ConsumableVirtualMemory**, the smallest amount you can request is 1 MB. If no units are specified, megabytes

are assumed. Resources defined here that are not in the **SCHEDULE_BY_RESOURCES** list in the global configuration file will not affect the scheduling of the job.

When resource usage and resource submission is enforced, either the **resources** or **node_resources** keyword must specify requirements for the resources defined in the **ENFORCE_RESOURCE_USAGE** keyword.

Both **resources** and **node_resources** can be specified in a single job step provided that the resources requested in each are different.

ConsumableCpus cannot be used in the **node_resources** list when the **rset** keyword is also specified in the job command file.

Default value: If the **node_resources** keyword is not specified in the job step, then the **default_node_resources** (if any) defined in the administration file for the class will be used for each task of the job step.

node_topology

Indicates that the scheduler must select the nodes to run the job step based on some common topology grouping. If no value is specified in the job command file, the default value is none, which means topology is not a consideration when the scheduler selects nodes to run the job step, and the job step could span multiple groups.

Job steps that specify **node_topology** must specify the **node** keyword. If **blocking**, **task_geometry**, or **host_file** is used, then the submission will fail.

By specifying **node_topology**, the value for **node_usage** is assumed to be **not_shared**. If **node_usage=shared** and **node_topology** are both specified for a job step, the request will automatically be changed to **node_usage=not_shared**.

Syntax:

```
node_topology = [none | island]
```

where:

none Specifies that topology is not a consideration when the scheduler selects nodes to run the job step. The nodes could come from any group or groups of machines.

island Specifies that the nodes will be selected to run job steps based on the island to which they belong. If the **island_count** keyword is not specified for the job step, then all nodes will be from the same island.

Default value: none.

node_usage

Specifies whether this job step shares nodes with other job steps.

This keyword is supported by the BACKFILL and API schedulers.

Syntax:

```
node_usage = shared | not_shared
```

where:

shared
Specifies that nodes can be shared with other tasks of other job steps.

not_shared
Specifies that nodes are not shared. No other job steps are scheduled on this node.

Default value: shared

nofile_limit

Specifies the hard limit, soft limit, or both for the number of open file descriptors that can be used by each process of the submitted job. This limit is a per process limit.

Syntax:

nofile_limit = *hardlimit*,*softlimit*

Default value: No default value is set.

Example:

```
nofile_limit = 1000,386
```

For additional information about limit keywords, see the following topics:
- "Syntax for limit keywords" on page 295
- "Using limit keywords" on page 94

notification

Specifies when the user specified in the **notify_user** keyword is sent mail.

Syntax:

notification = always|error|start|never|complete

where:

always
: Notify the user when the job begins, ends, or if it incurs error conditions.

error Notify the user only if the job fails.

start Notify the user only when the job begins.

never Never notify the user.

complete
: Notify the user only when the job ends.

Default value: complete

Examples:

- If you want to be notified with mail only when your job step completes, your notification keyword would be:

  ```
  notification = complete
  ```

- When a LoadLeveler job ends, you may receive mail notification indicating the job exit status. For example, you could get the following mail message:

  ```
  Your LoadLeveler job
  myjob1
  exited with status 4.
  ```

  The return code 4 is from the user's job. LoadLeveler retrieves the return code and returns it in the mail message, but it is not a LoadLeveler return code.

notify_user

Specifies the user to whom mail is sent based on the **notification** keyword.

Syntax:

notify_user = *userID*

Default value: The default is the submitting user at the submitting machine.

Example: If you are the job step owner but you want a coworker whose name and user ID is **bob**, to receive mail regarding the job step, your notify keyword would be:

```
notify_user = bob@mailserv.pok.ibm.com
```

nproc_limit

Specifies the hard limit, soft limit, or both for the number of processes that can be created for the real user ID of the submitted job. This limit is a per process limit.

Syntax:

nproc_limit = *hardlimit*,*softlimit*

Default value: No default value is set.

Example:

```
nproc_limit = 288,200
```

For additional information about limit keywords, see the following topics:
- "Syntax for limit keywords" on page 295
- "Using limit keywords" on page 94

output

Specifies the name of the file to use as standard output (stdout) when your job step runs.

Syntax:

output = *filename*

Default value: If you do not specify this keyword, LoadLeveler uses the file **/dev/null**

Example:

```
output = out.$(jobid)
```

parallel_threads

Requests OpenMP thread-level binding.

This keyword assigns separate CPUs to individual threads of an OpenMP task. The CPUs assigned to threads are selected from the set of CPUs or cores assigned to the task. The CPUs for individual OpenMP threads of the tasks are selected based on the number of parallel threads in each task and set of CPUs or cores assigned to the task. The CPUs are assigned to the threads only when at least one CPU is available for each thread from the set of CPUs or cores assigned to the task. If cores are assigned to the task instead of CPUs, then parallel threads are assigned to the CPUs of the cores in a round-robin method until all threads are bound.

LoadLeveler uses the **parallel_threads** value to set the value for the OMP_NUM_THREADS OpenMP runtime environment variable for all job types. For serial jobs, LoadLeveler uses the **parallel_threads** value to set the GOMP_CPU_AFFINITY (GNU compiler), KMP_AFFINITY (Intel compiler) runtime environment variables and the **'parthds'** suboption of the XLSMPOPTS runtime environment variable. See C/C++ and Fortran compiler documentation for more details about the OpenMP runtime environment.

Syntax:

parallel_threads = *number*

where *number* specifies the number of parallel threads in each OpenMP task of the job.

Default value: No default value is set.

preferences

Specifies the characteristics that you prefer be available on the machine that executes the job steps. LoadLeveler attempts to run the job steps on machines that meet your preferences. If such a machine is not available, LoadLeveler will then assign machines that meet only your requirements.

The values you can specify in a **preferences** statement are the same values you can specify in a **requirements** statement, with the exception of the **Adapter** requirement.

Syntax:

preferences = *Boolean_expression*

where:

Default value: No default preferences are set.

Examples:

```
preferences = (Memory <=16) && (Arch == "R6000")
preferences = Memory >= 64
```

queue

Places one or more copies of the job step in the queue. This statement is required. The **queue** statement essentially marks the end of the job step. Note that you can specify statements between **queue** statements.

Syntax:

queue = [*number*]

where:

number
Is the number of serial job steps to generate from the prior job command file statements. A value may only be specified for steps with **job_type = serial**. Only the **#@queue** syntax (without a value) is permitted for steps with **job_type = bluegene**, **job_type = mpich**, or **job_type = parallel**.

Default value: The default value is 1.

Note: The specification of a number on the queue statement is limited to the job command file that contains only one queue statement.

Job command file keyword values specified prior to this **queue** statement must be inherited by all proceeding job steps. The following keywords are not permitted when using this feature because their values are not inherited:

- **bg_block**
- **bg_connectivity**
- **bg_requirements**
- **bg_rotate**
- **bg_shape**
- **bg_size**
- **blocking**
- **bulkxfer**
- **coschedule**
- **host_file**
- **network**
- **node**
- **step_resources**
- **task_geometry**

- **tasks_per_node**
- **total_tasks**

Example:

There are two ways to specify five identical steps for a job. Five separate queue statements can be specified for a serial job:

```
#@ queue
#@ queue
#@ queue
#@ queue
#@ queue
```

Or a single queue statement with a value of 5 may be specified:

```
#@ queue = 5
```

recurring

Indicates whether the job will be run in every occurrence of the reservation to which it is bound.

Syntax:

recurring = yes | no

This keyword applies to all steps of a job. When all steps of a job have terminated that had **recurring = yes** specified, all steps will be automatically bound to the next occurrence of the reservation that they are bound to and are scheduled to run again. If the steps are not bound to any reservation or the reservation has expired, the job will be removed from the system upon termination of all steps as though the **recurring** keyword had not been set.

If a step specifies both **recurring = yes** and **restart =yes**, it is not a conflict. The **recurring** keyword indicates that once the step has terminated, it will be able to run again in the next occurrence of the reservation to which it is bound. Termination could be the result of successful completion, cancellation by the job's owner or an administrator, or a vacate or rejection, or some other failure. The **restart** keyword allows the same occurrence of the step to be restarted in the event of a vacate or rejection. A step could be restarted several times, then finally run to completion, then be requeued as a recurring job to run again. Similarly, a job could be rejected up to the maximum allowed rejects for a job, be removed as a result, then requeued as a recurring job to run in the next occurrence of its reservation.

Note that to permanently remove a recurring job, the job must first be unbound from its reservation, then removed with the **llcancel** command. As long as a recurring job is still bound to a reservation, it will continue to be requeued upon termination to run in the next occurrence of that reservation.

Default value: no

requirements

Specifies the requirements which a machine in the LoadLeveler cluster must meet to execute any job steps. You can specify multiple requirements on a single requirements statement.

Syntax:

requirements = *Boolean_expression*

When strings are used as part of a Boolean expression that must be enclosed in double quotes. Sample requirement statements are included following the descriptions of the supported requirements, which are:

Adapter
Specifies the predefined type of network you want to use to run a parallel job step. In any new job command files you create, you should use the **network** keyword to request adapters and types of networks.

It is also the way to specify when running with the default LoadLeveler scheduler. When using the default scheduler, the **Adapter** requirement is specified as the physical name of the device, such as *en0*.

This keyword is supported by the LL_DEFAULT and BACKFILL schedulers.

Note that you cannot specify both the **Adapter** requirement and the **network** statement in a job command file.

For the BACKFILL scheduler you can use the predefined network types. The predefined network types are:

ethernet
Refers to Ethernet.

fddi Refers to Fiber Distributed Data Interface (FDDI).

tokenring
Refers to Token Ring.

fcs Refers to Fiber Channel Standards.

Note that LoadLeveler converts the network types to the **network** statement.

Arch
Specifies the machine architecture on which you want your job step to run. It describes the particular kind of platform for which your executable has been compiled.

Connectivity
Connectivity is the ratio of the number of active switch adapters on a node to the total number of switch adapters on the node. The value ranges from 0.0 (all switch adapters are down) to 1.0 (all switch adapters are active). A node with no switch adapters has a connectivity of 0.0 . Connectivity can be used in a **MACHPRIO** expression to favor nodes that do not have any down switch adapters or in a job **REQUIREMENTS** statement to require only nodes with a certain connectivity.

Disk
Specifies the amount of disk space in kilobytes you believe is required in the LoadLeveler **execute** directory to run the job step.

Note: The Disk variable in an expression associated with the **requirements** and **preferences** keywords are 64-bit integers.

Feature
Specifies the name of a feature defined on a machine where you want your job step to run. Be sure to specify a feature in the same way in which the feature is specified in the configuration file. To find out what features are available, use the **llstatus** command.

Island
Specifies the name of the island where you want your job step to run.

LargePageMemory
Specifies the amount, in megabytes, of Large Page memory required to run the job.

Note: The Memory variable in an expression associated with the **requirements** and **preferences** keywords are 64-bit integers.

LL_Version
Specifies the LoadLeveler version, in dotted decimal format, on which you want your job step to run. For example, LoadLeveler Version 5 Release 1 (with no modification levels) is written as 5.1.0.0.

Machine
Specifies the names of machines on which you want the job step to run. Be sure to specify a machine name in lower case and in the same way in which it is specified in the configuration.

MachineGroup
Specifies the name of the machine group where you want your job step to run.

Memory
Specifies the amount, in megabytes, of regular physical memory required in the machine where you want your job step to run.

Note: The Memory variable in an expression associated with the **requirements** and **preferences** keywords are 64-bit integers.

OpSys
Specifies the operating system on the machine where you want your job step to run. It describes the particular kind of platform for which your executable has been compiled.

Pool
Specifies the number of a pool where you want your job step to run.

Region
Specifies the name of the region where you want your job step to run.

SMT
Specifies the SMT state of the machines where the job request is to run. Accepted values are "Disabled" and "Enabled."

If "Enabled" is specified, only machines on which SMT is enabled are considered for the job request. If "Disabled" is specified, only machines on which either SMT is disabled or machines that do not support SMT are considered for the job request.

TotalMemory
Specifies the amount, in megabytes, of regular physical memory and Large Page memory required in the machine where you want your job step to run.

Note: The Memory variable in an expression associated with the **requirements** and **preferences** keywords are 64-bit integers.

Default value: No default requirements are set.

Examples:

- **Example 1:** To specify a memory requirement and a machine architecture requirement, enter:

```
requirements = (Memory >=16) && (Arch == "R6000")
```

- **Example 2:** To specify that your job requires multiple machines for a parallel job, enter:

```
requirements = (Machine == { "l16" "l15" "l10" })
```

- **Example 3:** You can set a machine equal to a job step name. This setting means that you want the job step to run on the same machine on which the previous job step ran. For example:

  requirements = (Machine == machine.*step_name***)**

  where *step_name* is a step name previously defined in the job command file. The use of **Machine == machine.***step_name* is limited to serial jobs.

  Example:

  ```
  # @ step_name    = step1
  # @ executable   = c1
  # @ output       = $(executable).$(jobid).$(step_name).out
  # @ queue
  # @ step_name    = step2
  # @ dependency   = (step1 == 0)
  # @ requirements = (Machine == machine.step1)
  # @ executable   = c2
  # @ output       = $(executable).$(jobid).$(step_name).out
  # @ queue
  ```

- **Example 4:** To specify a requirement for a specific pool number, enter:

  ```
  requirements = (Pool == 7)
  ```

- **Example 5:** To specify a requirement that the job runs on LoadLeveler Version 5 Release 1 or any follow-on release, enter:

  ```
  requirements = (LL_Version >= "5.1")
  ```

  Note that the statement **requirements = (LL_Version == "5.1")** matches only the value 5.1.0.0.

- **Example 6:** To specify the job runs if all switch connections are up, enter:

  ```
  # @ requirements = (Connectivity == 1.0)
  ```

  To specify the job runs if at least half of the switch connections are up, enter:

  ```
  # @ requirements = (Connectivity >= .5)
  ```

  To specify the job runs if there is at least some connectivity, enter:

  ```
  # @ requirements = (Connectivity > 0)
  ```

- **Example 7:** To specify a requirement for SMT-enabled machines, enter:

  ```
  # @ requirements = (SMT == "Enabled")
  ```

resources

Specifies quantities of the consumable resources consumed by each task of a job step. The resources may be machine resources or floating resources.

Syntax:

resources=*name(count) name(count) ... name(count)*

where *name(count)* is one of the following:

- An administrator defined name and count
- **ConsumableCpus***(count)*
- **ConsumableMemory***(count units)*
- **ConsumableVirtualMemory***(count units)*
- **ConsumableLargePageMemory***(count units)*

The count for each specified resource must be an integer greater than or equal to zero, except for the following instances in which the integer must be greater than zero:

- **ConsumableMemory**
- **ConsumableVirtualMemory**

- **ConsumableCpus** when the enforcement policy is *hard* or *soft*

ConsumableCpus can have a value of zero when the administrator has not requested that consumable resources be enforced, or when the enforcement policy is *shares*.

When you set **ConsumableCpus** to zero, the meaning varies depending on whether use is being enforced. With no enforcement, zero means the job is requesting a negligible amount of CPU. With an enforcement policy of *shares*, it means the job is requesting a tiny percentage of available shares.

If the count is not valid then LoadLeveler will issue a message and the job will not be submitted. The allowable units are those normally used with LoadLeveler data limits:

```
b  bytes
w  words     (4 bytes)
kb kilobytes (2**10 bytes)
kw kilowords (2**12 bytes)
mb megabytes (2**20 bytes)
mw megawords (2**22 bytes)
gb gigabytes (2**30 bytes)
gw gigawords (2**32 bytes)
tb terabytes (2**40 bytes)
tw terawords (2**42 bytes)
pb petabytes (2**50 bytes)
pw petawords (2**52 bytes)
eb exabytes  (2**60 bytes)
ew exawords  (2**62 bytes)
```

ConsumableMemory, **ConsumableVirtualMemory**, and **ConsumableLargePageMemory** values are stored in MB (megabytes) and rounded up. For **ConsumableMemory** or **ConsumableVirtualMemory** the smallest amount which you can request is 1 MB. If no units are specified, then megabytes are assumed. However, **image_size** units are in kilobytes. Resources defined here that are not in the **SCHEDULE_BY_RESOURCES** list in the global configuration file will not affect the scheduling of the job.

When resource usage and resource submission is enforced, the **resources** keyword must specify requirements for the resources defined in the **ENFORCE_RESOURCE_USAGE** keyword.

Default value: If the **resources** keyword is not specified in the job step, then the **default_resources** (if any) defined in the administration file for the class will be used for each task of the job step.

restart

Specifies whether LoadLeveler considers a job to be "restartable."

Syntax:

```
restart = yes|no
```

If **restart=yes**, and the job is vacated from its executing machine before completing, the central manager requeues the job. It can start running again when a machine on which it can run becomes available. If **restart=no**, a vacated job is canceled rather than requeued.

Note that jobs which are checkpointable (**checkpoint = yes | interval**) are always considered "restartable".

If **restart=no**, a vacated, user-hold, or system-hold job is canceled rather than requeued.

Default value: yes

restart_from_ckpt

Indicates whether a job step is to be restarted from a checkpoint file.

Restriction: This keyword is ignored by LoadLeveler for Linux.

Syntax:

```
restart_from_ckpt = yes | no
```

where:

yes Indicates LoadLeveler will restart the job step from the checkpoint files in the directory specified by the **ckpt_subdir** job command file keyword, which must be set when **restart_from_ckpt = yes**. .

If **ckpt_subdir** does not contain a fully qualified path name, the location of the file or directory will be determined by the default location, which is the current directory.

This value is valid only when a job is being restarted from a previous checkpoint.

no The job step will be started from the beginning, not from the checkpoint file.

Default value: no

If you specify a value that is not valid for this keyword, the system generates an error message and the job is not submitted.

Tips:

- If the value specified by the **restart_from_ckpt** keyword is not valid, an error message is generated and the job is not submitted.
- The **restart_from_ckpt = yes** keyword value is used only when a job is being restarted from a previous checkpoint.

restart_on_same_nodes

Indicates that a job step is to be restarted on the same set of nodes that it was run on previously. This keyword applies only to restarting a job step after a vacate (this condition is when the job step is terminated and then returned to the LoadLeveler job queue).

Syntax:

```
restart_on_same_nodes = yes | no
```

where:

yes Indicates that the job step is to be restarted on the same set of nodes on which it had run.

no Indicates that it is not required to restart a vacated job on the same nodes.

Default value: no

Tip: If the value specified by the **restart_on_same_nodes** keyword is not valid, an error message is generated and the job is not submitted.

rset

This keyword indicates that the job tasks need to be attached to RSets with CPUs selected by different LoadLeveler scheduling algorithms or RSets created by users.

Syntax:

rset = *value*

value can be a user-defined RSet name or the following keyword:

RSET_MCM_AFFINITY
Specifying this value requests affinity scheduling with memory affinity as a requirement, adapter affinity as a preference, and the task MCM allocation method set to accumulate. The affinity options may be changed from these defaults by using the **mcm_affinity_options** keyword.

When anything other than the **RSET_MCM_AFFINITY** is specified, LoadLeveler considers the value to be a user-defined RSet name and schedules the job to nodes with **RSET_SUPPORT** set to **RSET_USER_DEFINED**.

Default value: If **task_affinity** is specified, then the default is **rset = RSET_MCM_AFFINITY**; otherwise, no default is set.

Example:

```
rset = RSET_MCM_AFFINITY
```

This example shows how to request affinity scheduling for a job. To request processor-core affinity, the jobs needs to specify the **task_affinity** keyword.

shell

Specifies the name of the shell to use for the job step.

Syntax:

shell = *name*

Default value: If you do not specify a value for this keyword, LoadLeveler uses the shell used in the owner's password file entry. If none is specified, LoadLeveler uses `/bin/sh`

Example: If you want to use the Korn shell, the shell keyword would be:

```
shell = /bin/ksh
```

smt

Indicates the required simultaneous multithreading (SMT) state for the job step.

LoadLeveler can satisfy this job request on AIX by dynamically enabling SMT if a node has SMT disabled. Once the job completes, LoadLeveler will return SMT to its original state.

To enable or disable **smt**, you must specify **smt = yes** or **smt = no** in the job command file. You must also specify **job_type = parallel** and **node_usage = not_shared**. For example:

```
#@ job_type = parallel
#@ node_usage = not_shared
#@ smt = yes | no
```

If **smt = as_is** is specified, it is not necessary to specify **job_type** or **node_usage**.

Syntax:

smt = yes | no | as_is

where:

yes
The job step requires SMT to be enabled.

no The job step requires SMT to be disabled.

as_is
The SMT state will not be changed.

Default value: as_is.

Examples:

```
smt = yes
```

stack_limit

Specifies the hard limit, soft limit, or both limits for the size of the stack that is created.

Syntax:

stack_limit = *hardlimit*,*softlimit*

Default value: No default is set.

Example:

```
stack_limit = 120000,100000
```

Because no units have been specified in this example, LoadLeveler assumes that the figure represents a number of bytes.

For additional information about limit keywords, see the following topics:
- "Syntax for limit keywords" on page 295
- "Using limit keywords" on page 94

startdate

Specifies when you want to run the job step.

Syntax:

startdate = *date time*

date is expressed as *MM/DD/YYYY*, and *time* is expressed as *HH:mm(:ss)*.

Default value: If you do not specify a start date, LoadLeveler uses the current date and time.

Example: If you want the job to run on August 28th, 2010 at 1:30 PM, issue:

```
startdate = 08/28/2010 13:30
```

If you specify a start date that is in the future, your job is kept in the Deferred state until that start date.

step_name

Specifies the name of the job step. You can name the job step using any combination of letters, numbers, underscores (_) and periods (.). You cannot, however, name it T or F, or use a number in the first position of the step name. The step name you use must be unique and can be used only once.

Syntax:

step_name = *step_name*

Default value: If you do not specify a step name, by default the first job step is named the character string "0", the second is named the character string "1", and so on.

Example:

```
step_name = step_3
```

step_resources

Specifies quantities of the floating resources consumed by a job step. The

resources must be floating resources and be in the **SCHEDULE_BY_RESOURCES** list in the global configuration file.

Syntax:

```
step_resources = name(count) name(count)...name(count)
```

where: *name* is an administrator defined name and *count* is an integer greater than or equal to zero. This keyword is not inherited by other job steps.

Default value: No default value is set.

Example:

```
step_resources = Conslicense(4) ConsBW(10)
```

task_affinity

Requests affinity scheduling in an SMT environment.

This keyword accepts either **core** or **cpu** as values. Optionally, you can specify an integer quantity for **core** or **cpu** by specifying a quantity within parenthesis '(' and ')'. If not specified, the default quantity is one.

The job will be scheduled to run only to the machines having LoadLeveler **ConsumableCpus** and **rset_mcm_affinity** configurations. When **core** is specified, each task of the job is bound to run on as many processor cores as specified. If **cpu** is specified, each task of the job is bound to run on the specified number of logical CPUs selected from the same cores. The CPUs allocated to the tasks of a processor-core affinity job will not be shared by other tasks. The processor cores allocated to the tasks of a processor-core affinity job can only be shared by tasks from other jobs, but not by tasks from the same job. If **task_affinity** is specified without the **rset** job command file keyword, **rset** is set to **rset_mcm_affinity** and **mcm_affinity_options** is set to **"mcm_mem_pref mcm_accumulate mcm_sni_pref"** for an **sn_single** parallel job; otherwise, it is set to **"mcm_mem_pref mcm_accumulate mcm_sni_none"**.

In a multistep job command file, if you set **task_affinity** equal to a blank, no task affinity will be set for the job step. If no **task_affinity** is specified for this job step, it will inherit the **task_affinity** value from its previous step.

Syntax:

```
task_affinity = core[(number)] | cpu[(number)]
```

Default value: If **parallel_threads =** *number* is specified, then the default is **task_affinity = cpu(***number***)**; otherwise, no default is set.

Example:

```
1. task_affinity = core

2. task_affinity = core(2)
   cpus_per_core = 1

3. task_affinity = cpu(4)
   cpus_per_core = 1
```

task_geometry

The **task_geometry** keyword allows you to group tasks of a parallel job step to run together on the same node. Although **task_geometry** allows for a great deal of flexibility in how tasks are grouped, you cannot specify the particular nodes that these groups run on; the scheduler will decide which nodes will run the specified groupings.

This keyword is supported by the BACKFILL and API schedulers.

Syntax:

task_geometry={(*task id*,*task id*,...)(*task id*,*task id*, ...) ... **}**

Default value: No default value is set.

Example: A job with 6 tasks will run on 4 different nodes:

```
task_geometry={(0,1) (3) (5,4) (2)}
```

Each number in this example represents a task ID in a job, each set of parenthesis contains the task IDs assigned to one node. The entire range of tasks specified must begin with 0, and must be complete; no number can be skipped (the largest task id number should end up being the value that is one less than the total number of tasks). The entire statement following the keyword must be enclosed in braces, and each grouping of nodes must be enclosed in parenthesis. Commas can only appear between task IDs, and spaces can only appear between nodes and task IDs.

The **task_geometry** keyword cannot be specified under **any** of the following conditions:

- The step is serial.
- **job_type** is anything other than **parallel**
- Any of the following keywords are specified:
  - **tasks_per_node**
  - **total_tasks**
  - **node**
  - **blocking**
  - **endpoints**, where the value specified is not equal to 1.

For more information, see "Task-assignment considerations" on page 186.

tasks_per_node

Specifies the number of tasks of a parallel job you want to run per node. Use this keyword together with the **node** keyword. The value you specify on the **node** keyword can be a range or a single value. If the node keyword is not specified, then the default value is one node.

The maximum number of tasks a job step can request is limited by the **total_tasks** keyword in the administration file (provided this keyword is specified). That is, the maximum must be less than any **total_tasks** value specified in a user, group, or class stanza.

The value of the **tasks_per_node** keyword applies only to the job step in which you specify the keyword. (That is, this keyword is not inherited by other job steps.)

Also, you cannot specify both the **tasks_per_node** keyword and the **total_tasks** keyword within a job step.

This keyword is supported by the BACKFILL and API schedulers.

Syntax:

tasks_per_node = *number*

where *number* is the number of tasks you want to run per node.

Default value: The default is one task per node.

Example: To specify a range of seven to 14 nodes, with four tasks running on each node, enter the following:

```
node = 7,14
tasks_per_node = 4
```

This job step runs 28 to 56 tasks, depending on the number of nodes allocated to the job step.

total_tasks

Specifies the total number of tasks of a parallel job you want to run on all available nodes. Use this keyword together with the **node** keyword. The value you specify on the **node** keyword must be a single value rather than a range of values. If the node keyword is not specified, then the default value is one node.

The maximum number of tasks a job step can request is limited by the **total_tasks** keyword in the administration file (provided this keyword is specified). That is, the maximum must be less than any **total_tasks** value specified in a user, group, or class stanza. The value of the **total_tasks** keyword applies only to the job step in which you specify the keyword. (That is, this keyword is not inherited by other job steps.) Also, you cannot specify both the **total_tasks** keyword and the **tasks_per_node** keyword within a job step.

If you specify an unequal distribution of tasks per node, LoadLeveler allocates the tasks on the nodes in a round-robin fashion. For example, if you have three nodes and five tasks, two tasks run on the first two nodes and one task runs on the third node.

This keyword is supported by the BACKFILL and API schedulers.

Syntax:

total_tasks = *number*

where *number* is the total number of tasks you want to run.

Default value: No default is set.

Example: To run two tasks on each of 12 available nodes for a total of 24 tasks, enter the following:

```
node = 12
total_tasks = 24
```

trace

Requests tracing of a job step's lifecycle from its submission time.

For tracing to occur, the administrator must first set the **TRACE** configuration file keyword to allow job life cycle tracing in the cluster. After that, either this keyword can be set to **yes** in the job command file to trace a job, or as an alternative, the **LOADL_JOB_TRACE** environment variable can be set to **yes** to trace all jobs that are subsequently submitted.

To trace

Syntax:

trace = yes|no

where:

yes
Requests tracing records for the events in this job's life cycle.

no A request is not made for tracing records for the events in this job's life cycle.

Default value: no

user_priority

Sets the initial priority of your job step. Priority only affects your job steps. It orders job steps you submitted with respect to other job steps submitted by you, not with respect to job steps submitted by other users.

Syntax:

```
user_priority = number
```

where *number* is a number between 0 and 100, inclusive. A higher number indicates the job step will be selected before a job step with a lower number. Note that this keyword is not the UNIX *nice* priority.

This priority guarantees the order the jobs are considered for dispatch. It does not guarantee the order in which they will run.

Default value: The default priority is 50.

wall_clock_limit

Sets the hard limit, soft limit, or both limits for the elapsed time for which a job can run. In computing the elapsed time for a job, LoadLeveler considers the start time to be the time the job is dispatched.

If you are running the BACKFILL scheduler, you must either set a wall clock limit in the job command file or the administrator must define a wall clock limit value for the class to which a job is assigned. In most cases, this wall clock limit value should not be **unlimited**. For more information, see "Choosing a scheduler" on page 46.

Syntax:

```
wall_clock_limit = hardlimit,softlimit
```

An example is:

```
wall_clock_limit = 5:00,4:30
```

For additional information about limit keywords, see the following topics:
- "Syntax for limit keywords" on page 295
- "Using limit keywords" on page 94

Job command file variables

LoadLeveler has several variables you can use in a job command file. These variables are useful for distinguishing between output and error files.

You can refer to variables in mixed case, but you must specify them using the following syntax:

```
$(variable_name)
```

The following variables are available to you:

$(domain)
The domain of the host from which the job was submitted.

$(home)
The home directory for the user on the cluster selected to run the job. Since the user may differ from the submitting user when a remote cluster is selected to run the job and user mapping is used, so may the home directory differ.

$(host)
The hostname of the machine from which the job was submitted. In a job command file, the **$(host)** variable and the **$(hostname)** variable are equivalent.

$(jobid)
The sequential number assigned to this job by the Schedd daemon. The **$(jobid)** variable and the **$(cluster)** variable are equivalent.

$(schedd_host)
The hostname of the scheduling machine.

$(schedd_hostname)
The hostname and domain name of the scheduling machine.

$(stepid)
The sequential number assigned to this job step when multiple queue statements are used with the job command file. The **$(stepid)** variable and the **$(process)** variable are equivalent.

$(user)
The user name on the cluster selected to run the job. This might be a different user name than the user name who submitted the job. It is possible for the value of this variable to differ from the submitting user name when a remote cluster is selected to run the job and user name mapping is being used.

In addition, the following keywords are also available as variables. However, you must define them in the job command file. These keywords are described in detail in "Job command file keyword descriptions" on page 335.
- **$(executable)**
- **$(class)**
- **$(comment)**
- **$(job_name)**
- **$(step_name)**

Note that for the **$(comment)** variable, the keyword definition must be a single string with no blanks. Also, the **executable** statement automatically sets the **$(base_executable)** variable, which is the file name of the executable without the directory component. See Figure 15 on page 175 for an example of using the **$(base_executable)** variable.

Run-time environment variables

The following environment variables are set by LoadLeveler for all jobs. These environment variables are also set before running prolog and epilog programs. For more information on prolog and epilog programs, see "Writing prolog and epilog programs" on page 80.

LL_DSTG_IN_EXIT_CODE
The exit code from the inbound data staging program. LoadLeveler sets the environment variable to the value obtained from executing **WEXITSTATUS** on the status returned in the wait3 system call when the inbound data staging program terminates.

LL_ENERGY_TAG_NAME
Specifies the energy tag name associated with the job. LoadLeveler sets the environment variable before it executes the user-supplied frequency setting program.

LOADL_ACTIVE
The LoadLeveler version.

LOADL_BG_BLOCK
The name of the allocated block.

LOADL_BG_CONNECTIVITY
The connectivity per dimension of the allocated block.

LOADL_BG_IOLINKS
The I/O links in a midplane of the allocated block.

LOADL_BG_MPS
The midplanes of the allocated blocks.

LOADL_BG_SHAPE
The shape of the allocated block in the form *AxBxCxD* representing the number of midplanes

LOADL_BG_SIZE
The size of the allocated block in number of compute nodes.

LOADLBATCH
Set to **yes** to indicate that the job is running under LoadLeveler.

LOADL_CKPT_FILE
Identifies the directory and file name for checkpointing files. LoadLeveler will only set this environmental variable if checkpointing is enabled.

LOADL_HOSTFILE
Specifies the full path name of the file that contains the host names assigned to all the tasks of the step. This environment variable is available only when the **job_type** is set to **parallel** or **MPICH**. This file is created in the execute directory and is deleted once the step has completed. The host names are stored in the file as one host name per line. The base name of this file is **step_hosts.step_id**.

LOADL_JOB_NAME
The three part job identifier.

LOADL_PID
The process ID of the starter process.

LOADL_PROCESSOR_LIST
A blank-delimited list of hostnames allocated for the step. This environment variable is limited to 128 hostnames. If the value is greater than the 128 limit, the environment variable is not set.

LOADL_STARTD_PORT
The port number where the startd daemon runs.

LOADL_STEP_ACCT
The account number of the job step owner.

LOADL_STEP_ARGS
Any arguments passed by the job step.

LOADL_STEP_CLASS
The job class for serial jobs.

LOADL_STEP_COMMAND
The name of the executable (or the name of the job command file if the job command file is the executable).

LOADL_STEP_ERR
The file used for standard error messages (stderr).

LOADL_STEP_GROUP
The UNIX group name of the job step owner.

LOADL_STEP_ID
The job step ID.

LOADL_STEP_IN
The file used for standard input (stdin).

LOADL_STEP_INITDIR
The initial working directory.

LOADL_STEP_NAME
The name of the job step.

LOADL_STEP_NICE
The UNIX *nice* value of the job step. This value is determined by the **nice** keyword in the class stanza. For more information, see "Defining classes" on page 94.

LOADL_STEP_NUMBER
Contains the step number assigned for the running step. This number can serve as a task ID for multistep jobs that run the same application in each step to distinguish what work needs to be done by the application.

LOADL_STEP_OUT
The file used for standard output (stdout).

LOADL_STEP_OWNER
The job step owner.

LOADL_STEP_TYPE
The job type (SERIAL or PARALLEL)

LOADL_TOTAL_TASKS
Specifies the total number of tasks of the MPICH job step. This variable is available only when the **job_type** is set to MPICH.

RM_CPUTASK*n*
A comma-delimited list of CPU IDs allocated for task *n*.

RM_MEM_AFFINITY
This environment variable is set as a directive to the process manager of a parallel job. A value of yes indicates that a task must use the memory only located on the same non-uniform memory access (NUMA) nodes as the task's allocated CPUs. A value of no indicates that a task can use memory on all available NUMA nodes.

Job command file examples

These job command file examples may apply to your situation.

1. The following job command file creates an output file called **stance.78.out**, where `stance` is the host and 78 is the job ID:

```
# @ executable = my_job
# @ arguments  = 5
# @ output     = $(host).$(jobid).out
# @ queue
```

2. The following job command file creates an **output** file called **computel.step1.March05**:

```
# @ comment    = March05
# @ job_name   = computel
# @ step_name  = step1
# @ executable = my_job
# @ output     = $(job_name).$(step_name).$(comment)
# @ queue
```

3. For a Blue Gene/Q job using two midplanes, the values for the run-time environment variables would be set similar to the following:

```
LOADL_BG_BLOCK=LL11120510045401
LOADL_BG_SIZE=1024
LOADL_BG_SHAPE=1x1x1x2
LOADL_BG_CONNECTIVITY=Torus Torus Torus Torus
LOADL_BG_MPS=R01-M0,R01-M1
LOADL_BG_IOLINKS=R01-M0-N04-J06,R01-M0-N12-J06,R01-M0-N06-J06,
R01-M0-N02-J06,R01-M0-N08-J06,R01-M0-N14-J06,R01-M0-N00-J06,
R01-M0-N10-J06,R01-M0-N04-J11,R01-M0-N12-J11,R01-M0-N06-J11,
R01-M0-N02-J11,R01-M0-N08-J11,R01-M0-N14-J11,R01-M0-N00-J11,
R01-M0-N10-J11,R01-M1-N12-J06,R01-M1-N06-J06,R01-M1-N08-J06,
R01-M1-N10-J06,R01-M1-N00-J06,R01-M1-N04-J06,R01-M1-N02-J06,
R01-M1-N14-J06,R01-M1-N12-J11,R01-M1-N06-J11,R01-M1-N08-J11,
R01-M1-N10-J11,R01-M1-N00-J11,R01-M1-N04-J11,R01-M1-N02-J11,
R01-M1-N14-J11
```

For additional information, see “Examples: Job command files” on page 173.

Part 5. Appendixes

Appendix A. Troubleshooting LoadLeveler

LoadLeveler offers troubleshooting information to help you resolve problems.

This topic is divided into the following subtopics:

- "Frequently asked questions," which contains answers to questions frequently asked by LoadLeveler customers. This topic focuses on answers that may help you get out of problem situations. The questions and answers are organized into the following categories:
  - **LoadLeveler won't start.** See "Why won't LoadLeveler start?" on page 392 for more information.
  - **Jobs submitted to LoadLeveler do not run.** See "Why won't my job run?" on page 392 for more information.
  - **One or more of your machines goes down.** See "What happens to running jobs when a machine goes down?" on page 397 for more information.
  - **The central manager is not operating.** See "What happens if the central manager isn't operating?" on page 399 for more information.
  - **Resources need to be recovered from the Schedd machine**. See "How do I recover resources allocated by a Schedd machine?" on page 401 for more information.
  - **A core file needs to be found on Linux.** See "Why can't I find a core file on Linux?" on page 401 for more information.
  - **Inconsistencies are found in llfs output.** See "Why am I seeing inconsistencies in my llfs output?" on page 402 for more information.
  - **Configuration or administration file errors.** See "What happens if errors are found in my configuration or administration file?" on page 402 for more information.
  - **Miscellaneous questions.** See "Other questions" on page 403 for more information.
- "Troubleshooting in a multicluster environment" on page 405, which contains common questions and answers pertaining to operations within a multicluster environment.
- "Troubleshooting in a Blue Gene environment" on page 409, which contains common questions and answers pertaining to operations within a Blue Gene environment.
- "Helpful hints" on page 411, which contains tips on running LoadLeveler, including some productivity aids.
- "Getting help from IBM" on page 416, which tells you how to contact IBM for assistance.

It is helpful to create error logs when you are diagnosing a problem. See to "Configuring recording activity and log files" on page 52 for information on setting up error logs.

Frequently asked questions

This topic contains answers to questions frequently asked by LoadLeveler customers.

Why won't LoadLeveler start?

Follow these instructions if the **master** daemon will not run. If the **master** daemon will not run, go to the node where **LoadL_master** will not start and issue the following at the command line:

```
LoadL_master -t
```

This generates messages that might help to diagnose the problem. In addition, ensure the following are true:

1. The **Release** and **bin** directories are properly specified in the configuration files.
2. The administration file exists and is properly defined in the configuration file.
3. The central manager is correctly defined in the administration file.
4. The **log** directories are correctly defined in the configuration file.
5. The **spool**, **execute**, and **log** directories exist and permissions are set as follows:
   - The **spool** subdirectory is set to 700
   - The **execute** subdirectory is set to 1777
   - The **log** subdirectory is set to 775
6. The **LoadL_master** binary, in **/usr/lpp/LoadL/full/bin** for AIX or **/opt/ibmll/LoadL/full/bin** for Linux, is owned by **root** and has the setuid bit set.
7. The daemons are not already running. If they are already running, use the **ps** command to identify the processes, and then use the **kill** command to kill the daemons.
8. When cluster security services is enabled, all machines in the LoadLeveler cluster must list each other in their trusted hosts list for authentication.
9. When cluster security services is enabled, the **loadl** ID must be a member of the UNIX group identified by the SEC_SERVICES_GROUP configuration file keyword.

Note: LoadLeveler for Linux does not support cluster security services.

Why won't my job run?

If you submitted your job but it has not run, issue **llq -s** first to help diagnose the problem. If you need more help diagnosing the problem, refer to Table 58:

Table 58. Why your job might not be running

Why your job might not be running:	Possible solution
Job requires specific machine, operating system, or other resource.	Does the resource exist in the LoadLeveler cluster? If yes, wait until it becomes available.
Job requires specific job class	• Is the class defined in the administration file? Use **llclass** to determine this. If yes, • Is there a machine in the cluster that supports that class? If yes, you need to wait until the machine becomes available to run your job.
The maximum number of jobs are already running on all the eligible machines	Wait until one of the machines finishes a job before scheduling your job.

Table 58. Why your job might not be running (continued)

Why your job might not be running:	**Possible solution**

The start expression evaluates to false.	Examine the configuration files (both **LoadL_config** and **LoadL_config.local**) to determine the **START** control function expression used by LoadLeveler to start a job. As a problem determination measure, set the START and SUSPEND values, as shown in this example:
	`START: T`
	`SUSPEND: F`

A job step is running on the node that your job requires, and that job step's preemption rules list your job's class as one that cannot share the node	The running job step is in a job class for which an administrator has defined preemption rules through the PREEMPT_CLASS keyword. When your job step's class is listed in the ALL clause of that keyword, your job step must wait until the running job step finishes.
The priority of your job is lower than the priority of other jobs.	You cannot affect the system priority given to this job by the negotiator daemon but you can try to change your user priority to move this job ahead of other jobs you previously submitted using the **llprio** command.
The information the central manager has about machines and jobs may not be current.	Wait a few minutes for the central manager to be updated and then the job may be dispatched. This time limit (a few minutes) depends upon the polling frequency and polls per update set in the **LoadL_config** file. The default polling frequency is five seconds.
You do not have the same user ID on all the machines in the cluster.	To run jobs on any machine in the cluster, you have to have the same user ID and the same uid number on every machine in the pool. If you do not have a userid on one machine, your jobs will not be scheduled to that machine.
CtSec is enabled and the **.rhosts** file was not updated.	The **.rhosts** file should contain entries which specify all the host and user combinations allowed to submit jobs which will run as the local user. See "Steps for enabling CtSec services" on page 62 for more details.

Table 58. Why your job might not be running (continued)

Why your job might not be running:	Possible solution
Your job is not bound to a reservation under which nodes that your job requires to run are reserved	When an unbound job requires nodes that are reserved under a reservation, LoadLeveler will not start the job unless one of the following conditions is true: • The reservation was created with SHARED mode specified. If the reservation is using SHARED mode, your job will remain idle until the reservation state becomes Active_Shared. • The job's expected end time (current time plus the hard wall clock limit) indicates that the job will complete before the reservation starts. If neither condition is true, but you have the authority to use the reservation, you may use the **llbind** command to bind your job to the reservation. Otherwise, your unbound job will remain idle until the reservation completes or is canceled. To check the reservation's status and attributes, use the **llqres** command. To find out which reservations you may use, check with your LoadLeveler administrator, or enter the command **llqres -l** and check the names in the Users or Groups fields (under the Modification time field) in the output listing. If your user name or a group name to which you belong appears in these output fields, you are authorized to use the reservation.
Your job is bound to a reservation but the reservation is not active yet	LoadLeveler schedules bound job steps to run only when a reservation becomes active. Use the command **llq -l** to find the ID of the reservation to which the job is bound. Use the command **llqres -l** to find the start time of the reservation, and wait until that time to check the job status again.
Your job is bound to a reservation that does not reserve all of the resources that your job requires to run	If a bound job requires specific resources that are not available during the reservation period, LoadLeveler will not dispatch the job to run under the reservation. This situation can occur if the job requires one or more of the following: • Specific nodes that were not selected for the reservation. • More than the total number of reserved nodes. • Floating consumable resources, which cannot be reserved under a reservation. If the LoadLeveler cluster has the resources that the job requires, use the command **llbind -r**, which unbinds the job from the reservation.
Your job is bound to a reservation but the maximum number of jobs you may run has been reached already	If LoadLeveler detects that you currently are running the maximum number of jobs that you are allowed to run, it will not start your bound job even if the reservation is active.

Table 58. Why your job might not be running (continued)

Why your job might not be running:	Possible solution
Your job is bound to a reservation but the job's expected end time exceeds the reservation's end time	LoadLeveler will dispatch your job only if its expected end time (current time plus the hard wall clock limit) does not exceed the end time of the reservation, or if both of the following conditions are true: • This reservation is configured to allow jobs to continue running even when their expected end time exceeds the end of the reservation, and • The resources required to run your job are available. Otherwise, this bound job will remain idle until either: • The reservation completes or is canceled, or • You use the command **llbind -r**, which unbinds the job from the reservation.
Your job is bound to a reservation that does not exist	LoadLeveler puts your job in NotQueued state until the reservation is created. In that case, LoadLeveler will bind your job to the reservation. Otherwise, use the command **llbind -r** to unbind the job from the reservation.
Your job is being rejected because a communication error is occurring while attempting to start the job. (Hic_Comm_Error occurred during job start.)	Check for mail sent by LoadLeveler when a job is rejected. If the job is being rejected because of a communication error (Hic_Comm_Error), verify that you can connect to the node for which the error occurred over the interface (generally Ethernet adapter) configured for LoadLeveler. Also verify that the **Loadl_startd** daemon is running on that node.
Your flexible job could not find a corresponding reservation	A flexible job is attached to a flexible reservation ID. If there was an error and the flexible job was not able to be attached to a flexible reservation, the administrator can remove the flexible job from the queue. For more information about flexible reservations, see "Working with reservations" on page 203.

You can use the **llq** command to query the status of your job or the **llstatus** command to query the status of machines in the cluster. Refer to *LoadLeveler: Command and API Reference* for information on these commands.

Why won't my parallel job run?

If you submitted your parallel job but it has not run, issue **llq -s** first to help diagnose the problem. If issuing this command does not help, refer to Table 58 on page 392 and to Table 59 for more information:

Table 59. Why your job might not be running

Why your job might not be running:	Possible solution
The minimum number of processors requested by your job is not available.	Sufficient resources must be available. Specifying a smaller number of processors may help if your job can run with fewer resources.
The pool in your **requirements** statement specifies a pool that is either not valid or not available.	The specified pool must be valid and available.

Table 59. Why your job might not be running (continued)

Why your job might not be running:	Possible solution
The adapter specified in the **requirements** statement or the **network** statement identifies an adapter that is either not valid or not available.	The specified adapter must be valid and available. Use **llstatus -a** to check the status of the adapters in the system. Switch adapters that show a state of 'NOT READY' or '-1' should be reported to the LoadLeveler administrator. Switch adapters with a state of '-1' indicate that the machine those adapters are on could not be queried for status. If the network statement specifies **rcxtblocks**, only switch adapters can be used for the step.
Your user space job is requesting stripping (**sn_all**) and the nodes allocated to the job do not have the same number of networks.	A user space job requesting striping will be rejected if the allocated nodes do not have the same number of networks. If notification is enabled for the job, mail will be sent to the user indicating that the job was rejected because the network table for the job could not be loaded. Use the **llstatus –a** command to verify that all the nodes in the cluster have same number of networks configured. The system administrator must take appropriate actions to insure the number of networks is the same on all nodes that will run user space jobs.

Common set-up problems with parallel jobs

This topic presents a list of common problems found in setting up parallel jobs:

- If jobs appear to remain in a Pending or Starting state: check that the nameserver is consistent. Compare results of **host** *machine_name* and **host** *IP_address*
- For POE:
  - Specify the POE partition manager as the executable. Do *not* specify the parallel job as the executable.
  - Pass the parallel job as an argument to POE.
  - The parallel job must exist and must be specified as a full path name.
  - If the job runs in user space, specify the flag **-euilib us**.
  - Specify the correct adapter (when needed).
  - Specify a POE job only once in the job command file.
  - Compile only with the supported level of POE.
  - Specify only **parallel** as the *job_type*.

Why won't my checkpointed job restart?

If the job you submitted has the keyword **restart_from_ckpt = yes** and if the checkpoint file specified does not exist, the job will move to the Starting state and will then be removed from the queue.

A mail message will be generated indicating the checkpoint file does not exist and a message will also appear in the **SchedLog**. Verify the values of the **ckpt_subdir** keyword in the job command file to ensure that they resolve to the directory and file name of the desired checkpoint file.

Note: When a job is enabled for checkpoint, it is important to ensure the name of the checkpoint file is unique.

Why won't my submit-only job run?

If a job you submitted from a *submit-only* machine does not run, verify that you have defined the following statements in the machine stanza of the administration file of the submit-only machine:

```
submit_only = true
schedd_host = false
central_manager = false
```

Verify that another machine has set **schedd_host = true** and **schedd_runs_here = true**.

Why does a job stay in the Pending (or Starting) state?

If a job appears to stay in the Pending or Starting state, it is possible the job is continually being dispatched and rejected.

Check the setting of the **MAX_JOB_REJECT** keyword. If it is set to -1 the job will be rejected an unlimited number of times. Try resetting this keyword to a small number, such as 10. Also, check the setting of the **ACTION_ON_MAX_REJECT** keyword. These keywords are described in Chapter 10, "Configuration keyword reference," on page 231. The reason for rejecting a job is sent to the user in mail if notification is enabled for the job.

Run **llq –lx** command on the job which is stuck in starting (ST) or pending (VP, RP). At the bottom of the **llq –lx** output the nodes allocated to the job and status of the job on those nodes is displayed. Look for the first node whose status is either starting or pending. Generally the first node whose state is starting or pending failed to report status and caused the job to be stuck in starting or pending state. Verify the node whose state is starting or pending is up and the **LoadL_startd** daemon on that node is running.

What happens to running jobs when a machine goes down?

Both the startd daemon and the Schedd daemon maintain persistent states of all jobs. Both daemons use a specific protocol to ensure that the state of all jobs is consistent across LoadLeveler. In the event of a failure, the state can be recovered. Neither the Schedd nor the startd daemon discard the job state information until it is passed onto and accepted by another daemon in the process. Refer to Table 60 for more information.

Table 60. Troubleshooting running jobs when a machine goes down

If	Then
The network goes down but the machines are still running	If the network goes down but the machines are still running, when LoadLeveler is restarted, it looks for all jobs that were marked running when it went down. On the machine where the job is running, the startd daemon searches for the job and if it can verify that the job is still running, it continues to manage the job through completion. On the machine where Schedd is running, Schedd queues a transaction to the startd to reestablish the state of the job. This transaction stays queued until the state is established. Until that time, LoadLeveler assumes the state is the same as when the system went down.

Table 60. Troubleshooting running jobs when a machine goes down (continued)

If	Then
The network partitions or goes down.	All transactions are left queued until the recipient has acknowledged them. Critical transactions such as those between the Schedd and startd are recorded on disk. This ensures complete delivery of messages and prevents incorrect decisions based on incomplete state information.
The machine with startd goes down.	Because job state is maintained on disk in startd, when LoadLeveler is restarted it can forward correct status to the rest of LoadLeveler. In the case of total machine failure, this is usually "JOB VACATED", which causes the job to be restarted elsewhere. In the case that only LoadLeveler failed, it is often possible to "find" the job if it is still running and resume management of it. In this case LoadLeveler sends JOB RUNNING to the Schedd and central manager, thereby permitting the job to run to completion.
The central manager machine goes down.	All machines in the cluster send current status to the central manager on a regular basis. When the central manager restarts, it queries each machine that checks in, requesting the entire queue from each machine. Over the period of a few minutes the central manager restores itself to the state it was in before the failure. Each Schedd is responsible for maintaining the correct state of each job as it progressed while the central manager is down. Therefore, it is guaranteed that the central manager will correctly rebuild itself. All jobs started when the central manager was down will continue to run and complete normally with no loss of information. Users may continue to submit jobs. These new jobs will be forwarded correctly when the central manager is restarted.
The Schedd machine goes down	When Schedd starts up again, it reads the queue of jobs and for every job which was in some sort of active state (i.e. PENDING, STARTING, RUNNING), it queries the machine where it is marked active. The running machine is required to return current status of the job. If the job completed while Schedd was down, JOB COMPLETE is returned with exit status and accounting information. If the job is running, JOB RUNNING is returned. If the job was vacated, JOB VACATED is returned. Because these messages are left queued until delivery is confirmed, no job will be lost or incorrectly dispatched due to Schedd failure. During the time the Schedd is down, the central manager will not be able to start new jobs that were submitted to that Schedd. To recover the resources allocated to jobs scheduled by a Schedd machine, see “How do I recover resources allocated by a Schedd machine?” on page 401.
The **llsubmit** machine goes down	Schedd gets its own copy of the executable so it does not matter if the **llsubmit** machine goes down.

Why does llstatus indicate that a machine is down when llq indicates a job is running on the machine?

If a machine fails while a job is running on the machine, the central manager does not change the status of any job on the machine. When the machine comes back up the central manager will be updated.

Why won't my job run on a cluster with both AIX and Linux machines?

The default shell on Linux (in both Red Hat Enterprise Linux and SUSE Linux Enterprise Server) is **bash** and **bash** may not be available on AIX.

If a job step contains a **bash** script it will be rejected if it is run on an AIX node. The **ksh** is available on both AIX and Linux. You can specify which shell to use in the keyword **shell** in your job command file:

```
# @shell = /bin/ksh
```

Also, AIX and Linux are not binary compatible so jobs written in compiled languages such as C or FORTRAN must be compiled for the environment they will run on.

Why won't my jobs run that were directed to an idle pool?

To determine why a job that was directed to an idle pool, but did not run, first check the total number of jobs in the LoadLeveler queue. LoadLeveler can only process a specific number of jobs in the queue within a given amount of time. Some jobs are not considered for scheduling if they do not have the appropriate priority. In this case, despite the fact that a different pool is a requirement from the jobs, it is not a factor in LoadLeveler's scheduling consideration.

To direct LoadLeveler's attention to scheduling jobs based on a priority, a job priority must be set or LoadLeveler's system priority must be defined.

A job's priority can be set by the system administrator with the class it is associated with. See the class stanza in the **LoadL_admin** file. The stanza keyword for setting the job class priority is:

```
priority = number
```

where the larger the number, the higher the priority. An example of a class stanza with a high priority is:

```
priority = 1000
```

You can also use the **SYSPRIO** expression defined in **LoadL_config** as an alternative. An example **SYSPRIO** expression that emphasizes a job priority is:

```
SYSPRIO: (ClassSysprio * 100) - (QDate)
```

System administrators can experiment with either **priority** or **SYSPRIO** to determine the appropriate priority or expression for a specific LoadLeveler cluster. You only need to specify one or the other.

What happens if the central manager isn't operating?

In one of your machine stanzas specified in the administration file, you specified a machine to serve as the central manager.

It is possible for some problem to cause this central manager to become unusable such as network communication or software or hardware failures. In such cases, the other machines in the LoadLeveler cluster believe that the central manager machine is no longer operating. If you assigned one or more alternate central managers in the machine stanza, a new central manager will take control. The alternate central manager is chosen based upon the order in which its respective machine stanza appears in the administration file.

Once an alternate central manager takes control, it starts up its negotiator daemon and notifies all of the other machines in the LoadLeveler cluster that a new central manager has been selected. Figure 19 illustrates how a machine can become the alternate central manager.
The diagram illustrates that Machine Z is the primary central manager but

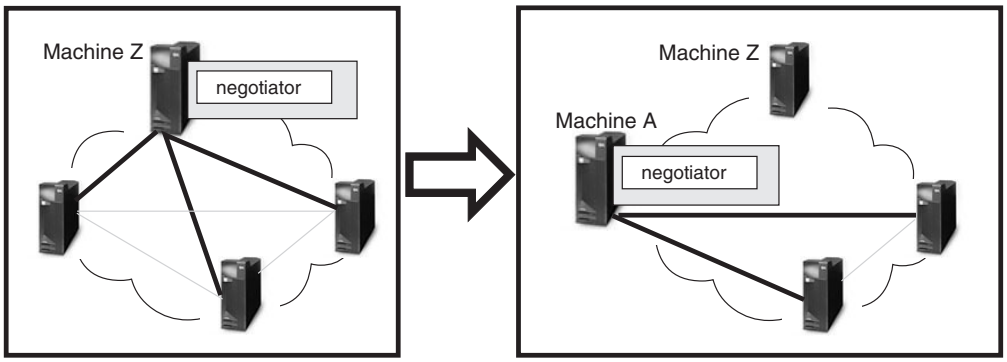

Figure 19. When the primary central manager is unavailable

Machine A took control of the LoadLeveler cluster by becoming the alternate central manager. Machine A remains in control as the alternate central manager until either:

- The primary central manager, Machine Z, resumes operation. In this case, Machine Z notifies Machine A that it is operating again and, therefore, Machine A terminates its negotiator daemon.
- Machine A also loses contact with the remaining machines in the pool. In this case, another machine authorized to serve as an alternate central manager takes control. Note that Machine A may remain as its own central manager.

Figure 20 illustrates how multiple central managers can function within the same LoadLeveler pool.

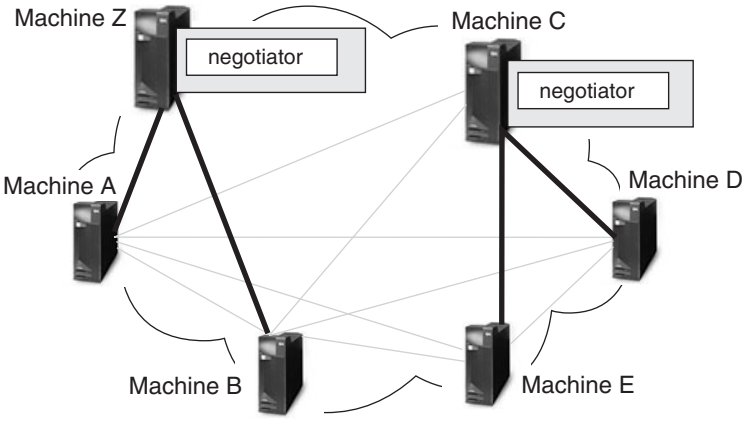

Figure 20. Multiple central managers

In this diagram, the primary central manager is serving Machines A and B. Due to some network failure, Machines C, D, and E have lost contact with the primary central manager machine and, therefore, Machine C which is authorized to serve as an alternate central manager, assumes that role. Machine C remains as the alternate central manager until either:

- The primary central manager is able to contact Machines C, D, and E. In this case, the primary central manager notifies the alternate central managers that it

is operating again and, therefore, Machine C terminates its negotiator daemon. The negotiator daemon running on the primary central manager machine is refreshed to discard any old job status information and to pick up the new job status information from the newly rejoined machines.

- Machine C loses contact with Machines D and E. In this case, if machine D or E is authorized to act as an alternate central manager, it assumes that role. Otherwise, there will be no central manager serving these machines. Note that Machine C remains as its own central manager.

While LoadLeveler can handle this situation of two concurrent central managers without any loss of integrity, some installations may find administering it somewhat confusing. To avoid any confusion, you should specify all primary and alternate central managers on the same LAN segment.

For information on selecting alternate central managers, refer to "Defining machines" on page 89.

How do I recover resources allocated by a Schedd machine?

If a node running the Schedd daemon fails, resources allocated to jobs scheduled by this Schedd cannot be freed up until you restart the Schedd.

Administrators must do the following to enable the recovery of Schedd resources:

1. Recognize that a node running the Schedd daemon is down and will be down long enough such that it is necessary for you to recover the Schedd resources.
2. Add the statement **schedd_fenced=true** to the machine stanza of the failed node. This statement specifies that the central manager ignores connections from the Schedd daemon running on this machine, and prevents conflicts from arising when a Schedd machine is restarted while **llmovespool** is running.
3. Reconfigure the central manager node so that it recognizes the "fenced" Schedd daemon. From the central manager machine issue **llctl reconfig**.
4. Issue the **llmovespool -d** *spool_directory* **-h** *target_schedd_hostname* command to move a job queue database to another Schedd within a local cluster.
5. Remove all files in the LoadLeveler spool directory of the failed node. Once the failed node is working again, you can remove the **schedd_fenced=true** statement.

For more information, see the **ll_move_spool** subroutine in *LoadLeveler: Command and API Reference* and the "Procedure for recovering a job spool" on page 164.

Why can't I find a core file on Linux?

On Linux, when a LoadLeveler daemon terminates abnormally a core file is not generated. Why?

Although a LoadLeveler daemon begins its existence as a root process, it uses the system functions **seteuid()** and **setegid()** to switch to effective user ID of **loadl** and effective group ID of **loadl** immediately after startup if the file **/etc/LoadL.cfg** is not defined. If this file is defined, the user ID associated with the **LoadLUserid** keyword and the group ID associated with the **LoadLGroupid** keyword are used instead of the default **loadl** user and group IDs.

On Linux systems, unless the default kernel runtime behavior is modified, the standard kernel action for a process that has successfully invoked **seteuid()** and **setegid()** to have a different effective user ID and effective group ID is not to

dump a core file. So, if you want Linux to create a core file when a LoadLeveler daemon terminates abnormally use one of the following commands to change the default kernel core file creation behavior.

If you want core files to be owned by the current user, use the command:

```
sysctl -w fs.suid_dumpable=1
```

If you want core files to be readable by root only, use the command:

```
sysctl -w fs.suid_dumpable=2
```

Why am I seeing inconsistencies in my llfs output?

Generally, the sum of the used shares by all users and the sum of the used shares by all LoadLeveler groups should have similar values.

If there is a large (much larger than the number of entries in the **llfs** output) and increasing difference between the two sums, the cluster should be checked to make sure that all nodes have the same clock time. Large time differences among nodes in a cluster could lead to errors in the **llfs** output.

Why don't I see my job when I issue the llq command?

By default, the **llq** command requires validation of the user ID that issues the command.

A user ID must exist on the central manager machine in order for LoadLeveler to validate the user ID. If the user ID does not exist on the central manager machine, then the validation will fail and the **llq** command will not return job data. User ID validation can be disabled by setting the **LoadL_config** file keyword **CM_CHECK_USERID** to **false**. Note that when user ID validation is disabled, any user can query data for any job.

What happens if errors are found in my configuration or administration file?

When errors are found during administration file processing, processing continues in nearly all cases. Because processing continues, it is possible that even though the administration file has been read, it might not resemble the intended configuration. This is especially true in cases where opening and closing braces are mismatched. It is possible for the parser to interpret stanzas as substanzas of another substanza, and when this happens, those stanzas are effectively ignored. Consider these cases:

- A machine stanza is interpreted to be a substanza within a class stanza. Because class stanzas only support substanzas of the user type, the machine substanza is completely ignored.
- A user stanza is incorrectly interpreted to be a substanza within a class. Although this is valid, that user stanza will not exist on its own, but will instead be part of the class stanza, which was not the administrator's intent.

It is difficult to determine whether an error in the administration file will completely change the meaning of the file or if the error will effect only a single keyword value. Because it is not necessarily desirable to shutdown LoadLeveler daemons and commands for every possible error, and because the behavior should be consistent, processing will continue. It is important for the administrator to be aware of this behavior and to investigate and repair any configuration errors reported by the **llctl start** command (see the **llctl** command in *LoadLeveler: Command and API Reference* for more information).

Use the **llctl ckconfig** command to display a list of configuration errors (see the **llctl** command in *LoadLeveler: Command and API Reference* for more information).

Why is my flexible reservation not activated?

If your flexible reservation has not been activated, check the flexible job by issuing the **llq -s** command to see the reason. Make sure there are resources available for the job to be scheduled.

Why was my energy aware job rejected?

If your energy aware job was rejected, it could be because the energy policy tag does not exist in the database. If you were trying to use an energy policy tag that was generated by another user, which is permissible, that tag must be valid in the system. LoadLeveler will reject the job if the tag does not exist.

For example, for user Tom to use the energy policy tag that was generated by John, the job command file would be:

```
# Tom.cmd
# @ energy_policy_tag = John.IO_job
```

In this example, John has to first generate the **IO_job** energy tag, otherwise this job will be rejected by LoadLeveler.

Your energy aware job might also be rejected because the required energy saving percentage cannot be satisfied.

For example, the maximum energy saving percentage of energy tag **long_running_job** is 20. Your job will be rejected if you required more than 20 percent energy savings in your job command file:

```
# User.cmd
# @ energy_policy_tag = long_running_job
# @ energy_saving_req = 21
```

Other questions

This topic contains answers to some miscellaneous questions asked by LoadLeveler customers.

Why do I have to setuid = 0?

The master daemon starts the **startd** daemon and the **startd** daemon starts the starter process.

The starter process runs the job. The job needs to be run by the userid of the submitter. You either have to have a separate master daemon running for every ID on the system or the master daemon has to be able to **su** to every userid and the only user ID that can **su** any other userid is **root**.

Why does LoadLeveler not execute my .profile or .login script?

When you submit a batch job to LoadLeveler, the operating system will execute your **.profile** script before executing the batch job if your login shell is the Korn shell.

On the other hand, if your login shell is the Bourne shell, on most operating systems (including AIX), the **.profile** script is not executed. Similarly, if your login

shell is the C shell then AIX will execute your **.login** script before executing your LoadLeveler batch job but some other variants of UNIX may not invoke this script.

The reason for this discrepancy is due to the interactions of the shells and the operating system. To understand the nature of the problem, examine the following C program that attempts to open a login Korn shell and execute the "ls" command:

```
#include <stdio.h>
main()
{
execl("/bin/ksh","-","-c","ls",NULL);
}
```

UNIX documentations in general (SunOS, HP-UX, AIX, IRIX) give the impression that if the second argument is "-" then you get a login shell regardless of whether the first argument is /bin/ksh or /bin/csh or /bin/sh. In practice, this is not the case. Whether you get a login shell or not is implementation dependent and varies depending upon the UNIX version you are using. On AIX you get a login shell for /bin/ksh and /bin/csh but not the Bourne shell.

If your login shell is the Bourne shell and you would like the operating system to execute your **.profile** script before starting your batch job, add the following statement to your job command file:

```
# @ shell = /bin/ksh
```

LoadLeveler will open a login Korn shell to start your batch job which may be a shell script of any type (Bourne shell, C shell, or Korn shell) or just a simple executable.

What happens when a mksysb is created when LoadLeveler is running jobs?

When you create a mksysb (an image of the currently installed operating system) at a time when LoadLeveler is running jobs, the state of the jobs is saved as part of the mksysb.

When the mksysb is restored on a node, those jobs will appear to be on the node, in the same state as when they were saved, even though the jobs are not actually there. To delete these phantom jobs, you must remove all files from the LoadLeveler **spool** and **execute** directories and then restart LoadLeveler.

What can I do when a reserved node is down?

If the reservation has not started yet, the node might become available before the reservation start time. If the node is still not available when the reservation starts, a LoadLeveler administrator may use the **llchres** command to remove the node and replace it with another.

How do I add or remove a node from the LoadLeveler administration file?

To add or remove a node from the LoadLeveler administration file, you must stop and restart LoadLeveler. Because stopping LoadLeveler will cause all running jobs to be vacated, you might want to drain the cluster before stopping LoadLeveler.

To add or remove nodes from the **LoadL_admin** file, do the following:

1. (**Optional**) Drain all nodes to allow all running jobs to complete by issuing the following command:

   ```
   llctl -g drain startd allclasses
   ```

2. Stop LoadLeveler by issuing the following command:

```
llctl -g stop
```

3. Add or remove nodes from the **LoadL_admin** file.
4. Start LoadLeveler by issuing the following command:

```
llctl -g start
```

Why does a job stay in the running state?

If a job appears to be stuck in the running state, it is possible that the node is not up or that a task has not completed.

Run **llq –lx** command on the job which is stuck in running state. At the bottom of the **llq –lx** output, the display shows the nodes allocated to the job and status of the job on those nodes. Look for the first node whose status is `running`. Generally the first node whose state is `running`, failed to report status and caused the job to be stuck in running state. Verify that the node whose state is `running` is actually up and that the **LoadL_startd** daemon on that node is running. If node is up and **LoadL_startd** is running then verify that all the tasks of the job are completed on that node.

Why is a job in the hold state?

If a job is in the hold state for an unknown reason, it is possible that LoadLeveler moved the job into that state.

LoadLeveler will move a job to hold state after the number of startup attempts reaches the value set in the configuration keyword **MAX_JOB_REJECT** and the configuration keyword **ACTION_ON_MAX_REJECT** is set to hold. The reason for rejecting a job is sent to the user in mail if notification is enabled for the job.

Why does the llstatus –a command show that adapters are NOT_READY?

If **llstatus –a** shows that adapters are **NOT_READY**, you must verify that the interfaces are running and that they are defined correctly.

Use the **ifconfig** and **ibstat** commands to verify that the interfaces are up and running. Reconfigure LoadLeveler using the **llctl –g reconfig** command. Check the PNSD log (**/tmp/serverlog**) to verify that the adapter status is UP.

Troubleshooting in a multicluster environment

This topic will help you troubleshoot your multicluster environment.

How do I determine if I am in a multicluster environment?

Use this procedure to determine whether your are functioning in a multicluster environment.

- Issue the **llstatus** command.
  - Output of command will display "Cluster name is *cluster_name*".

How do I determine how my multicluster environment is defined and what are the inbound and outbound hosts defined for each cluster?

Use this procedure to determine how your multicluster environment is defined and what are the inbound and outbound hosts defined for each cluster.

- Issue **llstatus -C** command.

- Output of command will display the local cluster's administration file cluster stanza information.

- Issue **llstatus -X all -C** command.
  - Output of command will display the administration file cluster stanza information for all clusters defined in the local cluster's configuration.

Why is my multicluster environment not enabled?

Use this procedure to determine why your multicluster environment is not enabled.

- Issue **llstatus -X all -C**.
  - The cluster stanzas defined for each cluster participating in the multicluster environment must have the same **outbound_hosts** and **inbound_hosts** defined.
  - Determine if any of the clusters are being started with **SCHEDD_STREAM_PORT** defined. The **inbound_schedd_port** keyword must be set for that cluster.
- Set the **D_MUSTER** debug flag for the **SCHEDD_DEBUG** configuration keyword on the machines defined as **inbound_hosts** and**outbound_hosts**, reconfigure LoadLeveler and examine the SchedLog on those machines for information about configuration errors.
- If the clusters are trying to enable OpenSSL, examine the SchedLog on the **inbound_hosts** and **outbound_hosts** for messages about SSL initialization errors and that multicluster is being disabled.

How do I find log messages from my multicluster-defined installation exits?

Use this procedure to determine how to find log messages from your multicluster-defined installation exits.

- Determine which machine is executing the installation exit.
  - For **CLUSTER_METRIC**:
    - If the user specifies the reserved word **any** as the **cluster_list** during job submission, the job is sent to the first outbound Schedd defined for the first configured remote cluster. The **CLUSTER_METRIC** is executed on this Schedd to determine where the job will be distributed. If this Schedd is not the **outbound_hosts schedd** for the assigned cluster, the job will be forwarded to the correct **outbound_hosts schedd**. If the user specifies a list of clusters as the **cluster_list** during job submission, the job is sent to the first outbound Schedd defined for the first specified remote cluster. The **CLUSTER_METRIC** is executed on this Schedd to determine where the job will be distributed. If this Schedd is not the **outbound_hosts schedd** for the assigned cluster, the job will be forwarded to the correct **outbound_hosts schedd**.
  - For **CLUSTER_USER_MAPPER**:
    - This installation exit is executed on the **inbound_hosts** of the local cluster when receiving a job submission, move job request or remote command.
  - For **CLUSTER_REMOTE_JOB_FILTER**:
    - This installation exit is executed on the **inbound_hosts** of the local cluster when receiving a job submission or move job request.

- Set the **D_MUSTER** debug flag for the **SCHEDD_DEBUG** configuration keyword on the machines defined as **inbound_hosts** and **outbound_hosts**, reconfigure LoadLeveler and examine the SchedLog on those machines for information about configuration errors.

Why won't my remote job be submitted or moved?

Use this procedure to determine why your remote job is not submitted or moved.

- Determine if the remote job filter has changed the number of steps within the job.
  - If the local submission filter on the submitting cluster has added or deleted steps from the original user's job command file, the remote job filter must add or delete the same number of steps. The job command file statements returned by the remote job filter must contain the same number of steps as the job object received from the sending cluster.
- Determine if the job failed the assigned cluster's include and exclude rules for the cluster and/or class stanzas.
  - If the assigned cluster has **CLUSTER_USER_MAPPING** enabled, the mapped user ID is applied to the rules.
- Issue **llq** to determine if the job being moved has all of its steps in an idle-like state.
  - The **llmovejob** command should fail and report this situation.
- Issue **llq -x -d** *job_id* to determine if the job being moved has a job command file associated with it.
  - A job cannot be moved that was not submitted while in the multicluster environment.
- See "How do I find log messages from my multicluster-defined installation exits?" on page 406 to determine if an installation exit has returned an error.
- Determine that the file system in the assigned cluster has the desired availability and permissions.
  - User may be mapped to another user ID thus another **$HOME**.
  - User needs to have **initialdir** available.
  - **cluster_input_file** and **cluster_output_file** need requested file locations to be available.
  - If clusters share a common file system, users requesting **cluster_input_file** and **cluster_output_file** may have their remote location files removed if a local job is moved to another cluster. During a **llmovejob** operation, the files are copied from the remote location to the remote location instead of from the local location to the remote location. LoadLeveler only knows that the job being moved has access to the remote location because they were copied during the local submission. After the **llmovejob** is complete, LoadLeveler removes the files from the local cluster in the remote location, thus removing the files just copied.
- Determine if the job is an interactive jobs.
  - Interactive jobs may not be submitted to remote clusters.
- If the **llsubmit** or **llmovejob** command times out while waiting for a response from the remote cluster, LoadLeveler is not able to determine if the command was successful and it is recommended that the user issue **llq** to the remote cluster to determine if the job was submitted or moved.

Why did the CLUSTER_REMOTE_JOB_FILTER not update the job with all of the statements I defined?

Use this procedure to determine why the **CLUSTER_REMOTE_JOB_FILTER** did not update the job with all of the statements that you defined.

- See the CLUSTER_REMOTE_JOB_FILTERCLUSTER_REMOTE_JOB_FILTER configuration file keyword description for a list of keywords that are not changed by the filter.

How do I find my remote job?

Use this procedure to find your remote job.

- Capture the **stdout** of the **llsubmit** and **llmovejob** commands to see the **outbound_hosts** machine assigned to the job, the **inbound_hosts** machine assigned to the job, the cluster assigned to the job, and the job identifier assigned to the job.
  - The Schedd host represented in the job identifier for remote jobs does not represent the managing Schedd of the job. It represents the Schedd that assigned the job number.
- Issue the **llq -X all** command and search for the desired job identifier.
- Check for pertinent mail messages.
  - If a job has been moved by an administrator, the submitting user will receive mail notification.
  - The job may have completed already. If the user has **notify_user** and **notification** set, mail will indicate job status.

Why won't my remote job run?

Use this procedure to determine why your remote job will not run. If the remote job has been received by the central manager of the remote cluster:

- Follow the troubleshooting tips for local jobs in "Why won't my job run?" on page 392 or "Why won't my parallel job run?" on page 395.
- Use the information from the **llsubmit** and **llq** commands to determine the machines that have processed the job. Examine the Schedd logs on those machines for information relating to the specific job.
- Capture the stdout of the **llsubmit** and **llmovejob** commands to see the **outbound_hosts** machine assigned to the job, the **inbound_hosts** machine assigned to the job, the cluster assigned to the job, and the job identifier assigned to the job.

  Note: The Schedd host represented in the job identifier for remote jobs does not represent the managing Schedd of the job. It represents the Schedd that assigned the job number.
- Issue **llq -X remote_cluster -l job_ID**.
- Check for the multicluster environment related keywords (see the **llq** command for detailed data descriptions):
  - Scheduling Cluster - what cluster is the job running in.
  - Submitting Cluster - what cluster was the job submitted from
  - Sending Cluster - during move job what cluster did the job come from
  - Requested Cluster - cluster_list specified by user.
  - Schedd History - history of managing Schedds
  - Outbound Schedds - history of outbound Schedds

  - Submitting User - user name that the job was submitted under
- Check for pertinent mail messages.

Why does llq -X all show no jobs running when there are jobs running?

Use this procedure to determine why **llq -X all** shows no jobs running when there are jobs running.

- When not using **CLUSTER_USER_MAPPER**, check that the user's uid are the same between the local cluster and remote cluster.

Troubleshooting adapter availability

If the user space adapters are down, or are in error, or they are seen as Ethernet adapters in the **llstatus -a** or **llrstatus -a** command, check the following:

- Run the **ifconfig** command to check adapter availability.
- Check to see if the PNSD daemon is up.
- Check the PNSD log to see if there are any errors.
- Ping the user space adapter addresses between machines.
- Set the **D_ADAPTERS** debug flag for Startd, reconfigure LoadLeveler and run the **grep** command for PNSD in the Startd log to see the error messages.

Troubleshooting in a Blue Gene environment

This topic will help you troubleshoot your Blue Gene environment.

For information on preparing your job for submission to the Blue Gene system, see "Submitting and monitoring Blue Gene jobs" on page 221.

Why do all of my Blue Gene jobs fail even though llstatus shows that Blue Gene is present?

Use this procedure to determine why all of your Blue Gene jobs fail even though **llstatus** shows that Blue Gene is present.

LoadLeveler queries Blue Gene information through the Blue Gene Scheduler APIs. If LoadLeveler receives all requested information from the Blue Gene Scheduler APIs without any problems, the **llstatus** output will indicate Blue Gene is present. Note, however, that other Blue Gene factors can cause jobs to fail.

In the event that the Blue Gene system is not completely ready to run jobs (for example, if the **bgmaster** status shows that some of the Blue Gene daemons are not running), then **runjob** jobs might not be able to run successfully. To determine if that is the case, run a **runjob** job outside of LoadLeveler and make sure that it can run to completion without problems. Sometimes file system failures can also cause all **runjob** jobs to fail. Contact the Blue Gene system administrator to resolve the problem when a **runjob** job fails outside of LoadLeveler. After those problems are resolved, run the jobs through LoadLeveler again.

Why does llstatus show that Blue Gene is absent?

Use this procedure to determine why **llstatus** shows that Blue Gene is absent.

LoadLeveler must load some of the Blue Gene libraries dynamically before it tries to call the Blue Gene Scheduler APIs. LoadLeveler will indicate that `Blue Gene is absent` in the **llstatus** output in the following situations:

- If the environment variables needed to use the Blue Gene Scheduler APIs and to run a **runjob** job are not properly set
- If the Blue Gene libraries cannot be loaded
- If the Blue Gene database cannot be accessed

LoadLeveler jobs will not be able to run in these situations. The LoadLeveler central manager log will contain error messages indicating what kind of failures occurred. Contact the Blue Gene administrator to resolve the problems in the Blue Gene environment. After all Blue Gene issues have been resolved, LoadLeveler might need to be restarted. The **llstatus** command must show `Blue Gene is present` before LoadLeveler jobs can run again.

Why did my Blue Gene job fail when the job was submitted to a remote cluster?

Use this procedure to determine why your Blue Gene job failed when the job was submitted to a remote cluster.

If Blue Gene jobs are submitted to a remote cluster where the version of the Blue Gene software is not compatible with the version installed on the submitting cluster, the executable statement cannot be used to specify the **runjob** program. The following job command file example shows how to invoke the **runjob** program without using the executable statement.

```
#!/bin/bash#
# @ comment = "Script to invoke runjob"
# @ error = $(job_name).$(jobid).out
# @ output = $(job_name).$(jobid).out
# @ wall_clock_limit = 00:20:00
# @ job_type = bluegene
# @ bg_size = 32
# @ queue
runjob --exe myexe --verbose 1 --args "myargs"
```

Why does llmkres or llchres return "Insufficient resources to meet the request" for a Blue Gene reservation when resources appear to be available?

If it appears that a Blue Gene reservation does not have enough resources, there are several simple items that you can check to determine if there is a real problem.

There are many reasons why resources are not available for a reservation request. These reasons include but are not limited to:

- The resources are in use
- The resources are not included in the job class or cluster
- The available resources are not the right type

In the case of a Blue Gene reservation, be aware that the number of I/O nodes in a midplane affect the smallest block size that can be supported by the midplane. If a midplane does not have enough I/O nodes, a reservation request asking for a smaller block than it can support will not get satisfied by the resources on the midplane.

Helpful hints

This topic contains tips on running LoadLeveler, including some productivity aids.

Scaling considerations

If you are running LoadLeveler on a large number of nodes (128 or more) consider the following recommendations:

- To reduce network traffic, and jitter from LoadLeveler daemons:
  - Set **POLLS_PER_UPDATE**, **POLLING_FREQUENCY**, **JOB_ACCT_Q_POLICY**, and **JOB_LIMIT_POLICY** such that **POLLING_FREQUENCY * POLLS_PER_UPDATE = JOB_ACCT_Q_POLICY = JOB_LIMIT_POLICY**. This reduces the number of timer threads required by LoadLeveler.
  - Set **UPDATE_ON_POLL_INTERVAL_ONLY = TRUE**. This limits how often Startd daemons send machine updates.
  - The following settings are recommended to reduce jitter and network traffic for installations with long-running parallel workloads. These settings reduce how often monitoring activity takes place and how often Startd daemons send updates to other LoadLeveler daemons.

    POLLING_FREQUENCY >= 300

    POLLS_PER_UPDATE = 1

    JOB_ACCT_Q_POLICY >= 300

    JOB_LIMIT_POLICY >= 300
- If your installation's mix of jobs includes a high percentage of parallel jobs requiring many nodes, specify **schedd_host=yes** in the machine stanza of each Schedd machine. The Schedd daemons must communicate with hundreds of startd daemons every time a job runs. You can distribute this communication by activating many Schedd daemons. Typically, the number of Schedd machines in a LoadLeveler cluster ranges from 2 to 10, depending on the mix of workload and number of jobs in the system.
- If your installation allows jobs to be submitted from machines running the Schedd daemon, you should consider avoiding “Schedd affinity” by specifying **SCHEDD_SUBMIT_AFFINITY=FALSE** in the LoadLeveler configuration file. By default, the **llsubmit** command submits a job to the machine where the command was invoked provided the Schedd daemon is running on the machine. (This is called Schedd affinity.)
- You can decrease the amount of time the negotiator daemon spends running negotiation loops by increasing the **NEGOTIATOR_INTERVAL** and the **NEGOTIATOR_CYCLE_DELAY**. For example, set **NEGOTIATOR_INTERVAL** to 600, and set **NEGOTIATOR_CYCLE_DELAY** to 30.
- Make sure the machine update interval is not too short by setting the **MACHINE_UPDATE_INTERVAL** to a value larger than three times the polling interval (**POLLS_PER_UPDATE*POLLING_FREQUENCY**). This prevents the negotiator from prematurely marking a machine as “down” or prematurely cancelling jobs.
- In a large LoadLeveler cluster, issuing the **llctl** command with **-g** can take minutes to complete. To speed this up, set up a working collective containing the machines in the cluster and use the **xdsh** command; for example, **xdsh llctl reconfig**. This command also allows you to limit your operation to a subset of machines by defining other working collectives.
- To reduce the cost of file I/O on execute nodes, the execute directory should be located on a local disk (if available) or in RAM disk.

Hints for running jobs

The following subtopics provide some helpful hints that are useful for running jobs.

Determining when your job started and stopped

By reading the notification mail you receive after submitting a job, you can determine the time the job was submitted, started, and stopped.

Suppose you submit a job and receive the following mail when the job finishes:

```
Submitted at: Mon Jan 14 11:40:41 2008
Started   at: Mon Jan 14 11:45:00 2008
Exited    at: Mon Jan 14 12:49:10 2008

Real Time:   0 01:08:29
Job Step User Time:   0 00:30:15
Job Step System Time:   0 00:12:55
Total Job Step Time:   0 00:43:10

Starter User Time:   0 00:00:00
Starter System Time:   0 00:00:00
Total Starter Time:   0 00:00:00
```

This mail tells you the following:

Submitted at
The time you issued the **llsubmit** command.

Started at
The time the starter process executed the job.

Exited at
The actual time your job completed.

Real Time
The wall clock time from submit to completion.

Job Step User Time
The CPU time the job consumed executing in user space.

Job Step System Time
The CPU time the system (AIX) consumed on behalf of the job.

Total Job Step Time
The sum of the **Job Step User Time** and **Job Step System Time** fields.

Starter User Time
The CPU time consumed by the LoadLeveler starter process for this job, executing in user space. Time consumed by the starter process is the only LoadLeveler overhead which can be directly attributed to a user's job.

Starter System Time
The CPU time the system (AIX) consumed on behalf of the LoadLeveler starter process running for this job.

Total Starter Time
The sum of the **Starter User Time** and **Starter System Time** fields.

You can also get the starting time by issuing **llsummary -l -x** and then issuing **awk /Date|Event/** against the resulting file. For this to work, you must have **ACCT = A_ON A_DETAIL** set in the **LoadL_config** file.

Running jobs at a specific time of day

Using a machine's local configuration file, you can set up the machine to run jobs at a certain time of day (sometimes called an *execution window*).

The following coding in the local configuration file runs jobs between 5:00 PM and 8:00 AM daily, and suspends jobs the rest of the day:

```
START: (tm_hour >= 1700) || (tm_hour <= 0800)
SUSPEND: (tm_hour > 0800)  && (tm_hour < 1700)
CONTINUE: (tm_hour >= 1700) || (tm_hour <= 0800)
```

Controlling the mix of idle and running jobs

The following keywords determine the mix of idle and running jobs for a user or group. These keywords, which are described in detail in "Defining users" on page 102, are:

maxqueued
: Controls the number of jobs in any of these states: Idle, Pending, Starting, Running, Preempt Pending, Preempted, Resume Pending, and Checkpointing.

maxjobs
: Controls the number of jobs in any of these states: Running, Pending, or Starting; thus it controls a subset of what **maxqueued** controls. The **maxjobs** keyword effectively controls the number of jobs in the Running state, since Pending and Starting are usually temporary states.

  Note: The **maxjobs** keyword can also be configured in the class stanza to limit the total number of running job steps of a particular class.

maxidle
: Controls the number of jobs in any of these states: Idle, Pending, or Starting; thus it controls a subset of what **maxqueued** controls. The **maxidle** keyword effectively controls the number of jobs in the Idle state, since Pending and Starting are usually temporary states.

Administrators can restrict the number of queued, idle, and running job steps on a per-class, per-user basis. The LoadLeveler administrator specifies the per-class, per-user constraints in the **LoadL_admin** file using user substanzas within each class stanza. For more information about substanzas, see "Defining user substanzas in class stanzas" on page 99.

What happens when you submit a job

This is what happens when you submit a job.

For a user's job to be allowed into the job queue and then dispatched:

- The total of other jobs (in the Idle, Pending, Starting, and Running states) for that user must be less than the **maxqueued** value for that user.
- The total idle jobs (those in the Idle, Pending, and Starting states) must be less than the **maxidle** value for the user.
- Constraints on the group's jobs and the user's jobs belonging to a particular class are considered.

Also, if the number of jobs exceeds the value specified by any of these **max** keywords, the job being considered is placed in the Not Queued state until one of the other jobs changes state. If the user is at the **maxqueued** limit, a job must complete, be canceled, or be held before the new job can enter the queue. If the user is at the **maxidle** limit, a job must start running, be canceled, or be held before the new job can enter the queue.

Once a job is in the queue, the job is not taken out of queue unless the user places a hold on the job, the job completes, or the job is canceled. This even applies to a job which is rejected or vacated and returned to the queue in the Idle state. (An exception to this, when you are running the default LoadLeveler scheduler, is parallel jobs which do not accumulate sufficient machines in a given time period. These jobs are moved to the Deferred state, meaning they must vie for the queue when their Deferred period expires.)

Once a job is in the queue, the job will run unless the **maxjobs** limit for the user is at a maximum.

Note the following restrictions for using these keywords:

- If **maxqueued** is greater than (**maxjobs** + **maxidle**), the **maxqueued** value will never be reached.
- If either **maxjobs** or **maxidle** is greater than **maxqueued**, then **maxqueued** will be the only restriction in effect, since **maxjobs** and **maxidle** will never be reached.

Sending output from several job steps to one output file

You can use dependencies in your job command file to send the output from many job steps to the same output file. For example:

```
# @ step_name = step1
# @ executable = ssba.job
# @ output = ssba.tmp
# @ ...
# @ queue
#
# @ step_name = append1
# @ dependency = (step1 != CC_REMOVED)
# @ executable = append.ksh
# @ output = /dev/null
# @ queue
# @
# @ step_name = step2
# @ dependency = (append1 == 0)
# @ executable = ssba.job
# @ output = ssba.tmp
# @ ...
# @ queue
# @
# @ step_name = append2
# @ dependency = (step2 != CC_REMOVED)
# @ executable = append.ksh
# @ output = /dev/null
# @ queue
#
# ...
```

Then, the file **append.ksh** could contain the line **cat ssba.tmp >> ssba.log**. All your output will reside in **ssba.log**. (Your dependencies can look for different return values, depending on what you need to accomplish.)

You can achieve the same result from within **ssba.job** by appending your output to an output file rather than writing it to **stdout**. Then your output statement for each step would be **/dev/null** and you wouldn't need the append steps.

Hints for using machines

The following subtopics provide some helpful hints for using machines.

Setting up a single machine to have multiple job classes

You can define a machine to have multiple job classes which are active at different times. For example, suppose you want a machine to run jobs of Class A any time, and you want the same machine to run Class B jobs between 6 p.m. and 8 a.m.

You can combine the **Class** keyword with a user-defined macro (called **Off_shift** in this example).

For example:

```
Off_Shift = ((tm_hour >= 18) || (tm_hour < 8))
```

Then define your START statement:

```
START : (Class == "A") || ((Class == "B") && $(Off_Shift))
```

Make sure you have the parenthesis around the **Off_Shift** macro, since the logical OR has a lower precedence than the logical AND in the START statement.

Also, to take weekends into account, code the following statements. Remember that Saturday is day 6 and Sunday is day 0.

```
Off_Shift = ((tm_wday == 6) || (tm_wday == 0) || (tm_hour >=18) \
|| (tm_hour < 8))

Prime_Shift = ((tm_wday != 6) && (tm_wday != 0) && (tm_hour >= 8) \
&& (tm_hour < 18))
```

Reporting the load average on machines

You can use the **/usr/bin/rup** command to report the load average on a machine.

The **rup** *machine_name* command gives you a report that looks similar to the following:

```
localhost    up 23 days, 10:25,    load average: 1.72, 1.05, 1.17
```

You can use this command to report the load average of your local machine or of remote machines. Another command, **/usr/bin/uptime**, returns the load average information for only your local host.

History files and Schedd

The **Schedd** daemon writes to the spool/history file only when a job is completed or removed. Therefore, you can delete the history file and restart **Schedd** even when some jobs are scheduled to run on other hosts.

However, you should clean up the **spool/LlObject.Job.***, **spool/LlObject.Step.*** and **spool/jobnnnnnn.*** files only when no jobs are being scheduled on the machine.

You should not delete these files if there are any jobs in the job queue that are being scheduled from this machine (for example, jobs with names such as *thismachine.clusterno.jobno*).

For fair share scheduling, **Schedd** daemons store historic CPU data for users and groups when their jobs terminate. Usually, a LoadLeveler cluster has more than one **Schedd** daemon. Each **Schedd** daemon only saves its own portion of the historic CPU data. Files in the directory specified by the **SPOOL** keyword on each Schedd machine that are named **LlObject.FairShare.*** contain the historic CPU data.

Note: Similarly for reservations, the **Schedd** daemon writes to the **reservation_history** file as each occurrence of each reservation completes. You should only clean up files named **llObject.Reservation.*** when no reservations are being managed by the machine.

Getting help from IBM

Should you require help from IBM in resolving a LoadLeveler problem, you can get assistance by calling IBM Support.

Before you call, be sure you have the following information:

1. Your access code (customer number).
2. The LoadLeveler product number.
3. The name and version of the operating system you are using.
4. A telephone number where you can be reached.

In addition, issue the following command:

```
llctl version
```

This command will provide you with code level information. Provide this information to the IBM representative.

The number for IBM support in the United States is 1-800-IBM-4YOU (426-4968).

The Facsimile number is 800-2IBM-FAX (2426-329).

Appendix B. LoadLeveler port usage

This topic describes LoadLeveler port usage.

A **port number** is an integer that specifies the port to use to connect to the specified daemon. For most ports used by LoadLeveler, you can define the port numbers in the configuration file or the **/etc/services** file or you can accept the defaults. LoadLeveler first looks in the configuration file for these port numbers. If LoadLeveler does not find the value in the configuration file, it looks in the **/etc/services** file. If the value is not found in this file, the default is used.

There are two exceptions to this rule:

1. The **LoadL_master_config** service cannot be specified in the LoadLeveler configuration data. This special port is used by LoadLeveler to retrieve configuration data from LoadLeveler nodes configured in the master configuration file as **LoadLConfigHost**. This port number can be specified in **/etc/services**.
2. Port numbers used by **sshd** daemons started by LoadLeveler for interactive jobs cannot be specified in **/etc/services**. A range of port numbers are used by LoadLeveler when starting an **sshd** daemon for an interactive job. This range can be configured using the **SSHD_PORTS** configuration keyword.

Note: See Table 61 on page 418 for the configuration file keywords associated with the port numbers.

The first column on each line in Table 61 on page 418 represents the name of a service. In most cases, these services are also the names of daemons with the following exceptions:

- **LoadL_negotiator_collector** is the service name for a second stream port that is used by the **LoadL_negotiator** daemon.
- **LoadL_schedd_status** is the service name for a second stream port used by the **LoadL_schedd** daemon.
- **LoadL_master_config** is the service name for a second stream port used by a **LoadL_master** daemon which is a configuration server when the database option is being used.

For each LoadLeveler service definition shown in Table 61 on page 418, the following information is shown:

Service name
Specifies the service name. The service names shown are examples of how the names might appear in the **/etc/services** file.

Port number
Specifies the port number used for the service.

Protocol name
Specifies the transport protocol used for the service.

Source port range
A range of port numbers used on either the client side or daemon (server) side of the service.

Required or optional
Whether or not the service is required.

Description/associated keywords
A short description of the service along with its associated configuration file keyword or keywords.

Table 61. LoadLeveler default port usage

Service name	Port number	Protocol name	Source port range[1]	Required or optional	Description/associated keywords
LoadL_master	9616	tcp	LB	Required	Master port number for stream port Keyword: **MASTER_STREAM_PORT** (tcp)
	9617	udp	LB	Required	Master port number for dgram port Keyword: **MASTER_DGRAM_PORT** (udp)
LoadL_master_config	9601	tcp	LB	Required	Master port number for configuration stream port Keyword: None. This port cannot be specified in the LoadLeveler configuration; only in **/etc/services**.
LoadL_negotiator	9614	tcp	LB	Required	Negotiator port number for stream port Keyword: **NEGOTIATOR_STREAM_PORT**
LoadL_negotiator_collector	9612	tcp	LB	Required	Second negotiator stream port Keyword: **CM_COLLECTOR_PORT** (tcp)
LoadL_region_mgr	9680	tcp	LB	Required	Region manger port number for stream port Keyword: **REGION_MGR_STREAM_PORT**
	9684	udp	LB	Required	Required adapter heartbeat port number Keyword: **ADAPTER_HEARTBEAT_PORT** (udp)

Table 61. LoadLeveler default port usage (continued)

Service name	Port number	Protocol name	Source port range[1]	Required or optional	Description/associated keywords
LoadL_resource_mgr	9618	tcp	LB	Required	Resource manager port number for stream port Keyword: **RESOURCE_MGR_STREAM_PORT**
	9619	udp	LB		Resource manager port number for dgram port Keyword: **RESOURCE_MGR_DGRAM_PORT** (udp)
LoadL_schedd	9605	tcp	LB	Required	Schedd port number for stream port Keyword: **SCHEDD_STREAM_PORT**
LoadL_schedd_status	9606	tcp	LB	Required	Schedd stream port for job status data Keyword: **SCHEDD_STATUS_PORT**
LoadL_startd	9611	tcp	LB	Required	Startd port number for stream port Keyword: **STARTD_STREAM_PORT**
	9615	udp	LB		Startd port number for dgram port Keyword: **STARTD_DGRAM_PORT** (udp)
N/A	9620-9629	tcp	LB	Required	Range of port numbers used by LoadLeveler when starting **sshd** daemons for interactive jobs Keyword: **SSHD_PORTS**

Note: [1]A value of LB indicates that the source port range value should be left blank. In other words, no source port range value should be specified.

For more information about configuration file keyword syntax and configuring the LoadLeveler environment, see the following:

- Chapter 4, “Configuring the LoadLeveler environment,” on page 39
- Chapter 10, “Configuration keyword reference,” on page 231

Accessibility features for LoadLeveler

Accessibility features help users who have a disability, such as restricted mobility or limited vision, to use information technology products successfully.

Accessibility features

The following list includes the major accessibility features in IBM LoadLeveler:

- Keyboard-only operation
- Interfaces that are commonly used by screen readers
- Keys that are discernible by touch but do not activate just by touching them
- Industry-standard devices for ports and connectors
- The attachment of alternative input and output devices

The *IBM Cluster information center*, and its related publications, are accessibility-enabled. The accessibility features of the information center are described in the IBM Cluster information center (http://publib.boulder.ibm.com/infocenter/clresctr/vxrx/topic/com.ibm.cluster.addinfo.doc/access.html).

Keyboard navigation

This product uses standard Microsoft Windows navigation keys.

IBM and accessibility

See the IBM Human Ability and Accessibility Center (http://www.ibm.com/able/) for more information about the commitment that IBM has to accessibility.

Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan, Ltd.
19-21, Nihonbashi-Hakozakicho, Chuo-ku
Tokyo 103-8510, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
Mail Station P300
2455 South Road,
Poughkeepsie, NY 12601-5400
USA

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample

programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

© your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at Copyright and trademark information (http://www.ibm.com/legal/copytrade.shtml).

Intel, Intel Inside (logos), MMX and Pentium are trademarks of Intel Corporation in the United States, other countries, or both.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft and Windows are trademarks of Microsoft Corporation in the United States, other countries, or both.

Red Hat, the Red Hat “Shadow Man” logo, and all Red Hat-based trademarks and logos are trademarks or registered trademarks of Red Hat, Inc., in the United States and other countries.

UNIX is a registered trademark of the Open Group in the United States and other countries.

Glossary

This glossary includes terms and definitions for LoadLeveler.

The following cross-references are used in this glossary:

- See refers you from a term to a preferred synonym, or from an acronym or abbreviation to the defined full form.
- See also refers you to a related or contrasting term.

To view glossaries for other IBM products, go to IBM Terminology (http://www-01.ibm.com/software/globalization/terminology/index.jsp).

A

AFS A distributed file system for large networks that is known for its ease of administration and expandability.

AIX A UNIX operating system developed by IBM that is designed and optimized to run on POWER microprocessor-based hardware such as servers, workstations, and blades.

authentication
The process of validating the identity of a user or server.

authorization
The process of obtaining permission to perform specific actions.

B

Berkeley Load Average
The average number of processes on the operating system's ready-to-run queue.

C

C language
A language used to develop application programs in compact, efficient code that can be run on different types of computers with minimal change.

client A system or process that is dependent on another system or process (usually called the *server*) to provide it with access to data, services, programs, or resources.

cluster
A collection of complete systems that work together to provide a single, unified computing capability.

D

daemon
A program that runs unattended to perform continuous or periodic functions, such as network control.

DCE See *Distributed Computing Environment*.

default
Pertaining to an attribute, value, or option that is assumed when none is explicitly specified.

DFS See *Distributed File System*.

Distributed Computing Environment (DCE)
In network computing, a set of services and tools that supports the creation, use, and maintenance of distributed applications across heterogeneous operating systems and networks.

Distributed File Service (DFS)
A component of a Distributed Computing Environment (DCE) that enables a single, integrated file system to be shared among all DCE users and host computers in a DCE cell. DFS prevents DCE users from simultaneously modifying the same information.

F

flexible job
A job step that is only used for scheduling resources for a flexible reservation.

flexible reservation
A reservation that starts as soon as resources are first available.

H

host A computer that is connected to a network and provides an access point to that network. The host can be a client, a server, or both a client and server simultaneously.

L

LAPI See *low-level application programming interface*.

low-level application programming interface (LAPI)
An IBM message-passing interface that implements a one-sided communication model.

M

MCM See *multiple chip module*.

memory affinity
A feature available in AIX to allocate memory attached to the same multiple chip module (MCM) on which the process runs. Memory affinity improves the performance of applications on IBM System p® servers.

menu A displayed list of items from which a user can make a selection.

Message Passing Interface (MPI)
A library specification for message passing. MPI is a standard application programming interface (API) that can be used with parallel applications and that uses the best features of a number of existing message-passing systems.

MPI See *Message Passing Interface*.

MPICH2
A portable implementation of the Message Passing Interface (MPI).

multiple chip module (MCM)
The fundamental, processor, building block of IBM System p servers.

N

network
In data communication, a configuration in which two or more locations are physically connected for the purpose of exchanging data.

Network File System (NFS)
A protocol, developed by Sun Microsystems, Incorporated, that enables a computer to access files over a network as if they were on its local disks.

NFS See *Network File System*.

node A computer location defined in a network.

nominal CPU frequency
The vendor specified frequency for the CPU.

P

parameter
A value or reference passed to a function, command, or program that serves as input or controls actions. The value is supplied by a user or by another program or process.

process
A separately executable unit of work.

R

RDMA
See *Remote Direct Memory Access*.

Remote Direct Memory Access (RDMA)
A communication technique in which data is transmitted from the memory of one computer to that of another without passing through a processor. RDMA accommodates increased network speeds.

resource set (RSet)
A data structure in AIX used to represent physical resources such as processors and memory. AIX uses resource sets to restrict a set of processes to a subset of the system's physical resources.

RSet See *resource set*.

S

S3 state
A power state where everything in the system is put into a low-power state except for memory.

server In a network, hardware or software that provides facilities to clients. Examples of a server are a file server, a printer server, or a mail server.

shell A software interface between users and an operating system. Shells generally fall into one of two categories: a command line shell, which provides a command line interface to the operating system; and a graphical shell, which provides a graphical user interface (GUI).

simultaneous multithreading (SMT)
Pertaining to a processor design that combines hardware multithreading with superscalar processor technology. Using

SMT, a single physical processor emulates multiple processors by enabling multiple threads to issue instructions simultaneously during each cycle.

SMT See *simultaneous multithreading*.

system administrator
The person who controls and manages a computer system.

T

TCP See *Transmission Control Protocol*.

Transmission Control Protocol (TCP)
A communication protocol used in the Internet and in any network that follows the Internet Engineering Task Force (IETF) standards for internetwork protocol. TCP provides a reliable host-to-host protocol in packet-switched communication networks and in interconnected systems of such networks.

U

UDP See *User Datagram Protocol*.

User Datagram Protocol (UDP)
An Internet protocol that provides unreliable, connectionless datagram service. It enables an application program on one machine or process to send a datagram to an application program on another machine or process.

W

workflow
An application that has been partitioned into a complex sequence of interdependent jobs is called a workflow. The execution of jobs in a workflow may depend on the success/failure of previously executed jobs. Some jobs in a workflow may need to use output of jobs that executed before them; jobs that are not interdependent may be able to execute in parallel.

workflow engine
A software tool that manages the execution of a workflow. By integrating with LoadLeveler, a workflow engine can benefit from features like flexible reservations and state change notifications to automate several workflows in parallel.

working directory
The active directory. When a file name is specified without a directory, the current directory is searched.

workstation
A configuration of input/output equipment at which an operator works. A workstation is a terminal or microcomputer at which a user can run applications and that is usually connected to a mainframe or a network.

Index

Special characters

Numerics

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Product Number: 5725-G01
5641-LL1
5641-LL3
5765-L50
5765-LLP

Printed in USA

SC23-6792-04