# LSF JobScheduler User's Guide

Fourth Edition, January 2000

Platform Computing Corporation

# LSF JobScheduler User's Guide

Copyright © 1994-2000 Platform Computing Corporation and its licensors
All rights reserved.

Printed in Canada

Although the information in this document has been carefully reviewed, Platform Computing Corporation does not warrant it to be free of errors or omissions. Platform Computing Corporation reserves the right to make corrections, updates, revisions or changes to the information in this document.

UNLESS PROVIDED OTHERWISE IN WRITING BY PLATFORM COMPUTING CORPORATION, THE PROGRAM DESCRIBED IN THIS DOCUMENT IS PROVIDED AS IS WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. IN NO EVENT WILL PLATFORM BE LIABLE TO ANYONE FOR SPECIAL, COLLATERAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES, INCLUDING ANY LOST PROFITS OR LOST SAVINGS, ARISING OUT OF THE USE OF OR INABILITY TO USE THIS PROGRAM.

## We'd Like to Hear from You

You can help us make this manual better by telling us what you think of the content, organization, and usefulness of the information. If you find an error, or just want to make a suggestion for improving this manual, please address your comments to doc@platform.com.

Your comments should pertain only to the LSF documentation. For product support, contact support@platform.com.

## Trademarks

LSF Base, LSF Batch, LSF JobScheduler, LSF MultiCluster, LSF Analyzer, LSF Make, LSF Parallel, Platform Computing, and the Platform Computing and LSF logos are trademarks or registered trademarks of Platform Computing Corporation and its licensors.

UNIX is a registered trademark of The Open Group.

Other products or services mentioned in this document are identified by the trademarks or service marks of their respective companies or organizations.

## Revision Information

This document has been revised as follows:

| Edition | Description |
|---|---|
| First | This document describes LSF JobScheduler 3.0. Based on the *LSF User's Guide*, Fourth Edition. |
| Second | Revised to reflect changes in LSF JobScheduler 3.1. |
| Third | Revised to reflect changes in LSF JobScheduler 3.2. |
| Fourth | Reprinted for LSF 4.0. |

# Contents

**Preface** . . . . . **ix**
Audience . . . . . ix
LSF Suite 3.2 . . . . . ix
Related Documents . . . . . x
Technical Assistance . . . . . xi

**1 - Introduction** . . . . . **1**
What is LSF JobScheduler? . . . . . 1
Structure of LSF JobScheduler . . . . . 2
LSF Cluster . . . . . 2
Jobs . . . . . 4
Job Groups . . . . . 4
Events . . . . . 4
Calendars . . . . . 5
Exceptions and Alarms . . . . . 5
Queues . . . . . 5
Inter-job Dependency . . . . . 6
Command Set and GUI Tools . . . . . 6
Job History . . . . . 7

**2 - Getting Started** . . . . . **9**
Getting to Know Your Cluster . . . . . 9
Displaying the Job Queues . . . . . 10
Displaying Host Load and Resource Information . . . . . 10
Displaying LSF JobScheduler Server Information . . . . . 11
Submitting a Job . . . . . 11
Creating a Calendar . . . . . 12
Creating a Calendar from the Command Line . . . . . 12
Creating a Calendar Using the LSF JobScheduler - Calendar GUI . . . . . 13
Displaying Calendars . . . . . 14
Displaying Calendars from the Command Line . . . . . 14
Displaying Calendars Using the LSF JobScheduler - Calendar GUI . . . . 15

Associating Jobs with Calendars . . . . . 15
Associating Jobs with Calendars from the Command Line . . . . . 15
Associating Jobs with Calendars Using the LSF JobScheduler - Job Submission GUI . . . . . 16
Deleting Jobs . . . . . 16
Associating Jobs with Other Jobs. . . . . 17
Associating Jobs with File Events . . . . . 17
Tracking Jobs . . . . . 18
Using LSF JobScheduler GUI Tools. . . . . 18

**3 - Events and Calendars . . . . . 21**
How Are Events Created? . . . . . 22
Event Status . . . . . 22
Built-in Events . . . . . 23
Time Events . . . . . 23
Job Events. . . . . 23
Job Group Events . . . . . 24
Job Exception Events . . . . . 25
External Events. . . . . 27
File Events . . . . . 27
User Events . . . . . 28
Viewing Events. . . . . 29
Calendars and Time Events . . . . . 31
What is a Calendar? . . . . . 31
Built-in Calendars and Reserved Names for Calendars . . . . . 31
System Calendars . . . . . 33
Using the LSF JobScheduler - Calendar GUI . . . . . 33
Calendar Expressions and the Command Line Interface . . . . . 37
Manipulating Calendars Using the Command Line Interface. . . . . 40
Time Events . . . . . 42

**4 - Resources . . . . . 45**
Introduction to Resources. . . . . 45
Load Indices. . . . . 47
Static Resources. . . . . 49
Shared Resources . . . . . 51
Resource Requirement Strings. . . . . 51
Selection String . . . . . 52
Order String. . . . . 54
Resource Usage String . . . . . 55

Job Resource Requirement Specification Examples. . . . . 55
Configuring Resource Requirements . . . . . 57
Remote Task List File . . . . . 57
Managing Your Task List . . . . . 57

**5 - Defining Jobs . . . . . 59**
Types of Jobs. . . . . 59
Job Attributes . . . . . 60
Job Status. . . . . 60
Creating a Simple Job . . . . . 63
Input and Output. . . . . 65
Host Selection . . . . . 67
Host Groups . . . . . 68
Host Preference. . . . . 69
Queue Selection . . . . . 69
Specifying the Default Queue. . . . . 70
Choosing a Queue. . . . . 70
Resource Requirements. . . . . 71
Pre-execution Commands. . . . . 72
File Transfer . . . . . 73
Grouping Jobs. . . . . 75
Job Group Status. . . . . 76
Creating a Job Group . . . . . 76
Submitting a Job under a Job Group . . . . . 79
Specifying Dependency Conditions . . . . . 80
Time Event Dependency . . . . . 80
Inter-job Dependencies . . . . . 83
File Event Dependency . . . . . 86
Job Exception Event Dependency . . . . . 89
User Event Dependency . . . . . 90
Combining Dependency Conditions . . . . . 91
Synchronizing Dependent Jobs . . . . . 91
Other Job Parameters. . . . . 91
Number of Processors for Parallel Jobs . . . . . 92
Start Time and Termination time . . . . . 92
Exclusive Job . . . . . 92
Ad-hoc Jobs . . . . . 93
Exception Handlers . . . . . 93

**6 - Managing Jobs and Schedules . . . . . 95**
Viewing Details of a Job or Job Group . . . . . 95
Viewing Job History . . . . . 99
Peeking at Job Output . . . . . 103
Modifying a Job . . . . . 104
Modifying a Job Group . . . . . 105
Deleting a Job or Job Group . . . . . 107
Deleting Jobs . . . . . 107
Deleting Job Groups . . . . . 108
Delayed Deletion of a Job . . . . . 109
Job Controls . . . . . 109
Terminating a Job . . . . . 109
Terminating a Group of Jobs . . . . . 110
Sending Arbitrary Signals to Jobs . . . . . 110
Suspending and Resuming Jobs . . . . . 111
Forcing a Job to Run . . . . . 112
Job Group Control . . . . . 112
Managing Schedules of Jobs . . . . . 113
System Status Monitoring . . . . . 114
Event View . . . . . 115
Host View . . . . . 115
Queue View . . . . . 117
Load View . . . . . 119

**7 - Exception Handling and Alarms . . . . . 123**
Exceptions . . . . . 123
Exception Handlers . . . . . 125
Using Exception Handlers . . . . . 126
Handling Failures with Built-in Exception Handlers . . . . . 126
Handling Failures with Recovery Jobs . . . . . 127
Setting Exception Handlers Using Command Line Interface . . . . . 129
Alarms . . . . . 130
How Are Alarms Generated . . . . . 130
Manipulating Alarms . . . . . 132

**Index . . . . . 135**

# Preface

## Audience

This guide is intended for users of LSF JobScheduler, and provides tutorial and reference information created for them. Users should be familiar with executing commands in a UNIX or Windows NT environment.

LSF JobScheduler administrators should also be familiar with the contents of the *LSF JobScheduler Administrator's Guide*.

## LSF Suite 3.2

LSF is a suite of workload management products including the following:

**LSF Batch** is a batch job processing system for distributed and heterogeneous environments, which ensures optimal resource sharing.

**LSF JobScheduler** is a distributed production job scheduler that integrates heterogeneous servers into a virtual mainframe or virtual supercomputer

**LSF MultiCluster** supports resource sharing among multiple clusters of computers using LSF products, while maintaining resource ownership and cluster autonomy.

**LSF Analyzer** is a graphical tool for comprehensive workload data analysis. It processes cluster-wide job logs from LSF Batch and LSF JobScheduler to produce statistical reports on the usage of system resources by users on different hosts through various queues.

**LSF Parallel** is a software product that manages parallel job execution in a production networked environment.

**LSF Make** is a distributed and parallel Make based on GNU Make that simultaneously dispatches tasks to multiple hosts.

**LSF Base** is the software upon which all the other LSF products are based. It includes the network servers (LIM and RES), the LSF API, and load sharing tools.

There are two editions of the LSF Suite:

### LSF Enterprise Edition

Platform's LSF Enterprise Edition provides a reliable, scalable means for organizations to schedule, analyze, and monitor their distributed workloads across heterogeneous UNIX and Windows NT computing environments. LSF Enterprise Edition includes all the features in LSF Standard Edition (LSF Base and LSF Batch), plus the benefits of LSF Analyzer and LSF MultiCluster.

### LSF Standard Edition

The foundation for all LSF products, Platform's Standard Edition consists of two products, LSF Base and LSF Batch. LSF Standard Edition offers users robust load sharing and sophisticated batch scheduling across distributed UNIX and Windows NT computing environments.

## Related Documents

The following guides are available from Platform Computing Corporation:

*LSF Installation Guide*
*LSF Batch Administrator's Guide*
*LSF Batch Administrator's Quick Reference*
*LSF Batch User's Guide*
*LSF Batch User's Quick Reference*
*LSF JobScheduler Administrator's Guide*
*LSF JobScheduler User's Guide*
*LSF Analyzer User's Guide*

*LSF Parallel User's Guide*
*LSF Programmer's Guide*

### Online Documentation

- Man pages (accessed with the `man` command) for all commands
- Online help available through the Help menu for the `xlsbatch`, `xbmod`, `xbsub`, `xbalarms`, `xbcal` and `xlsadmin` applications.

## Technical Assistance

If you need any technical assistance with LSF, please contact your reseller or Platform Computing's Technical Support Department at the following address:

LSF Technical Support
Platform Computing Corporation
3760 14th Avenue
Markham, Ontario
Canada L3R 3T7

Tel: +1 905 948 8448
Toll-free: 1-87PLATFORM (1-877-528-3676)
Fax: +1 905 948 9975
Electronic mail: *support@platform.com*

Please include the full name of your company.

You may find the answers you need from Platform Computing Corporation's home page on the World Wide Web. Point your browser to *www.platform.com*.

If you have any comments about this document, please send them to the attention of LSF Documentation at the address above, or send email to *doc@platform.com*.

# 1. Introduction

## What is LSF JobScheduler?

Production job scheduling has been an integral part of mainframe data processing operations for decades. With the emergence of distributed computing, along with UNIX and Windows NT workstations and file servers, system architecture has changed drastically, calling for a new approach to job scheduling.

LSF JobScheduler is part of Platform's Workload Management solutions. LSF JobScheduler centralizes and automates the scheduling of production workload in distributed UNIX and Windows NT environments. LSF JobScheduler integrates heterogeneous servers into a 'virtual mainframe' to deliver high availability, robustness, and ease-of-use. It provides the functions of traditional mainframe job scheduling with transparent operation across a network of heterogeneous UNIX and Windows NT systems.

With LSF JobScheduler, you can target jobs to specific servers, or you can let the system match the requirements of your jobs to the capabilities of your servers. LSF JobScheduler dynamically collects system load information about all aspects of computing resources including CPU, memory, I/O, disk space, and interactive activities. Jobs are dynamically scheduled to run on the most suitable servers available. LSF JobScheduler offers graphical tools in addition to the standard command line interface.

Some of the features of LSF JobScheduler are:

- a single system image for a heterogeneous network of computers
- dynamic and intelligent resource mapping and load balancing
- centralized monitoring of resource load information and job information

- calendar-driven job scheduling
- event-driven job scheduling
- site-defined event monitoring
- flexible exception handling and error recovery mechanism
- complex inter-job dependencies and job dependency-driven job scheduling
- high availability and fault tolerance
- flexible, hierarchical job grouping

# Structure of LSF JobScheduler

LSF JobScheduler consists of a master scheduler and a number of slave execution servers distributed across a cluster of computers. There is only one master scheduler in the whole cluster, and one slave execution server on each machine that runs jobs. The master scheduler (`mbatchd`) accepts jobs created by users and schedules jobs to run by slave execution servers on individual machines. A slave execution server (`sbatchd`) accepts jobs dispatched from the master scheduler and runs them on the local machine, controlling the execution according to job specifications from the master.

The components of LSF JobScheduler are shown in *Figure 1*.

## LSF Cluster

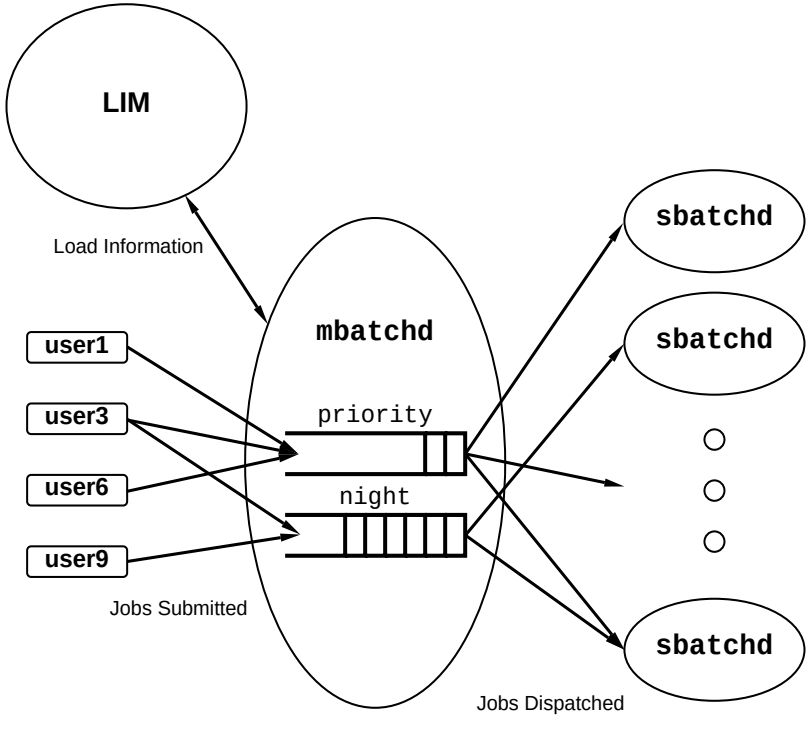
An LSF cluster is a group of computers that are configured to act as a single, integrated system for job scheduling. All machines configured into the cluster share resources transparently. An LSF cluster consists of one or more server hosts and zero or more client hosts. A client host is a machine that does not run user-submitted jobs, but allows users to submit, monitor, and control jobs running on LSF JobScheduler server hosts. A server host does everything a client host can do, and also runs user-submitted jobs .

One of the server hosts acts as the master for the cluster. It runs the master scheduler, `mbatchd`. Each server host runs a slave execution server, `sbatchd`, which manages jobs dispatched by the master scheduler. Each server host also runs a Load Information Manager daemon, LIM. It monitors the availability of resources and makes this information available to LSF JobScheduler and other LSF utilities.

Each cluster can have one or more LSF cluster administrators. An LSF cluster administrator is a user account that has permission to change the LSF JobScheduler configuration and perform other maintenance functions. The LSF cluster administrator has the authority to decide how the LSF JobScheduler cluster is configured.

The master scheduler maintains the status of all entities defined in the system including jobs, events, calendars, and queues.

Figure 1. Components of LSF JobScheduler



## Jobs

A *job* is a program or command that is scheduled to run in a specific environment. A job has many attributes specifying its scheduling and execution requirements. Job attributes are specified by the user who submits the job. LSF JobScheduler uses job attributes, system resource information, and configured scheduling policies to decide when, where, and how to run jobs. While each job is assigned a unique job identification number by the system, you can associate your own job names to make referencing easier.

## Job Groups

A job group is a container for jobs in much the same way that a directory is a container for files. When developing a complex schedule involving many jobs, it is useful to organize related jobs into groups so that it becomes easier to view and manipulate them. For example, a payroll application may have one group of jobs that calculates weekly payments, another job group for calculating monthly salaries, and a third job group that handles the salaries of part-time or contract employees. Users can view and operate on the job groups rather than looking at individual jobs.

## Events

An *event* is a change or occurrence in the system (such as the creation of a specific file, a tape drive becoming available, or a prior job completing successfully or at a particular time) that can be used to trigger jobs. LSF JobScheduler responds to the following types of events:

- **time events -** points of time (defined by calendars and time expressions) that can be used to trigger the scheduling of jobs.
- **job events -** the starting and completion of other jobs
- **job group events -** status condition changes of job groups
- **file events -** changes in a file's status
- **user events -** site-specific occurrences, such as a tape mount, defined by the LSF cluster administrator

- **exception events -** conditions raised in response to errors in the scheduling or execution of jobs

When defining a job, it is possible to specify any combination of events that must be satisfied before the job is considered eligible for execution.

## Calendars

A *calendar* consists of a sequence of days on which the calendar is considered active. A job is scheduled when the calendar is active and a time of day specification is met. Calendars are defined and manipulated independently of jobs so that multiple jobs can share the same calendar. Each user can maintain a private set of calendars, reference calendars of other users, or use the calendars configured into the system. A calendar can be modified after it has been created. Any new jobs associated with it will automatically run according to the new definition.

## Exceptions and Alarms

When managing critical jobs it is important to ensure that the jobs run properly. When problems are detected during the processing of the job, it becomes necessary to take some form of corrective action. LSF JobScheduler allows you to associate each job with one or more exception handlers which tell the system to watch for a particular type of error and take a specified action if it occurs. An exception condition represents a problem in processing a job. LSF JobScheduler can watch for several types of exception conditions during a job's life cycle.

An alarm specifies how a notification should be sent in the event of an exception.

## Queues

Production job scheduling provides efficient, timely execution of mission-critical jobs. When you submit a job, it is placed into a *queue*. The LSF JobScheduler system runs jobs from the queue based on the scheduled time and when the appropriate resources are available. Jobs from a queue can be dispatched to any server hosts in your cluster that are configured to run jobs for the queue.

A queue can be configured with many features that make your life easier. LSF JobScheduler allows you to define various types of services by configuring different

queues. For each queue, you can configure a set of parameters that customize job scheduling policies, job execution behaviour, and resource allocation constraints.

## Inter-job Dependency

LSF JobScheduler allows you to control a job's execution upon the completion, failure, or startup of other jobs. For example, you can configure the system to start several main processing jobs only after a data preparation job has completed, then you can start the post-processing job after all the main processing jobs are done. These jobs do not have to run on the same host.

## Command Set and GUI Tools

LSF JobScheduler provides a rich set of commands and GUI tools to define, monitor and manage the workload using any desktop as the system console. Typically, you define your calendars and jobs together with any interdependencies using the GUI tools `xbcal` and `xbsub`. Once these are set up, LSF JobScheduler will ensure that jobs are run according to the conditions and policies specified.

You can keep close track of your jobs with LSF JobScheduler using the GUI program `xlsjs`. As well as monitoring the status of jobs, the system allows you to perform various operations on them, including:

- terminating, suspending, and resuming each run of a job, as well as removing the entire job from the system
- inspecting the output of a running job
- looking at the history of a repetitive job for all its run instances
- changing any parameter of a job
- inquiring why a job has not been scheduled

LSF JobScheduler also comes with a comprehensive set of tools for monitoring your cluster. These tools allow you to view your cluster of resources from any host in the cluster so that you know the dynamic resource usage of all your machines.

## Job History

LSF JobScheduler maintains the full history data of all jobs. The history information tells you what has happened to your jobs.

# 2. Getting Started

You will most commonly use LSF JobScheduler to process periodic or repetitive jobs. To accomplish this, you need to first select or create a calendar. Next you must associate jobs with the calendar. Once you have performed these two tasks, LSF JobScheduler will automatically execute your jobs according to the calendar schedule and the resource requirements of your jobs.

You can also use LSF JobScheduler to process jobs based on the occurrence of some other events in the system, e.g. the creation (arrival) of a file.

LSF JobScheduler offers many utilities to check the status and progress of your jobs and calendars. You can set up alarms and exception handlers to handle job failure. Other utilities allow you to discover the resources available in the clusters on the network.

This chapter walks you through some of the most common LSF JobScheduler operations.

## Getting to Know Your Cluster

LSF JobScheduler enables you to submit and monitor your mission critical jobs in a heterogeneous network environment. It automatically balances the load on the hosts in your cluster so that your jobs run faster. Tools are provided to allow you to view your cluster resource and load information from any machine in your cluster. LSF JobScheduler gathers cluster resource information as well as system state information from all server hosts in the cluster. A variety of utilities are available to present such information in various formats.

## Displaying the Job Queues

Job queues represent different job scheduling and resource allocation services. All jobs submitted to a queue share the same policy. Each queue can be configured to use a subset of the servers in the LSF JobScheduler cluster.

The `bqueues` command lists the available queues.

```
% bqueues
QUEUE_NAME   PRIO   STATUS          NJOBS   PEND   RUN   SUSP
nightly      43     Open:Inactive   5       5      0     0
accounting   30     Open:Active     0       0      0     0
evaluation   20     Open:Active     0       0      0     0
```

## Displaying Host Load and Resource Information

The `lsload` command displays dynamic load information about LSF JobScheduler server hosts.

```
% lsload
HOST_NAME   status   r15s   r1m r15m   ut    pg   ls it   tmp   swp   mem
hostd       ok       0.4    0.3 0.1    14%   10   5  3    56M   203M  125M
hostb       ok       0.8    0.8 1.6    42%   7    4  0    10M   71M   43M
hosta       unavail
```

The hosts are ordered so that lightly loaded hosts are listed first. The status ‘unavail’ indicates that the Load Information Manager (LIM) on that host is not running.

The `lshosts` command shows the static resource information about LSF JobScheduler server hosts.

```
% lshosts
HOST_NAME  type   model   cpuf ncpus maxmem maxswp  server  RESOURCES
hosta      hppa   HP715   4    1     64M    128M    Yes     (hppa hpux)
hostb      sunsol sparc   3    1     96M    128M    Yes     (sun solaris)
hostd      SGI    R10000  10   8     512M   1024M   Yes     (sgi irix6)
```

Static resource information is either configured by your LSF administrator or automatically identified by LIM.

### Displaying LSF JobScheduler Server Information

The `bhosts` command displays information about LSF JobScheduler server hosts information.

```
% bhosts
HOST_NAME          STATUS    JL/U  MAX  NJOBS  RUN  SSUSP USUSP  RSV
hosta              ok        -     1    0      0    0     0      0
hostb              closed    -     1    1      1    0     0      0
hostd              ok        -     4    2      2    0     0      0
```

These fields display the status and job counters on the LSF JobScheduler servers. The status ‘closed’ indicates that the server will not accept any new jobs at the current time.

## Submitting a Job

To submit a job into LSF JobScheduler, use the `bsub` command or the `xbsub` graphical tool. If you submit a job that has no dependency conditions (such as time events and file events), then the job will be run once, at most, and will be removed from LSF JobScheduler after it finishes.

For example, you can submit the job “sleep_30” from the command line as follows:

```
% bsub sleep_30
Job <1234> is submitted to default queue <normal>.
```

Here, `1234` is the job ID assigned by LSF JobScheduler, and `normal` is the name of the default job queue defined by your LSF cluster administrator. There are a number of options you can specify to control the way in which your job should be scheduled.

See *Section 5, ‘Defining Jobs’, beginning on page 59*, for more information on controlling the scheduling of your job.

# Creating a Calendar

There are two possible types of calendars you can use to submit your job: a preconfigured system calendar, or a calendar you or another user created. See *'What is a Calendar?' on page 31* for more details.

## Creating a Calendar from the Command Line

The following example shows you how to create a calendar with the `bcadd` command.

```
% bcadd -d "Every Monday" -t "*:*:Monday" on_monday
Calendar <on_monday> is created.
```

This creates a calendar named 'on_monday' that is active every Monday. A calendar has a name, a calendar expression, and an optional description. The calendar name is case insensitive.

The optional description is a string of text declared with the `-d` option. If the string contains blanks or special characters, the entire string should be placed within quotes. This description string is displayed as part of the calendar information when you run bcal command with '-l' option.

```
% bcal -l
CALENDAR: on_monday
 -- Every Monday

OWNER  STATUS    CREATION_TIME              LAST_MODIFY_TIME
user1  inactive  Fri Nov 14 17:50:01 1997   -

CAL_EXPRESSION:  *:*:Monday
LAST_CAL_DAY: <Mon Nov 10 1997>
NEXT_CAL_DAY: <Mon Nov 17 1997>
```

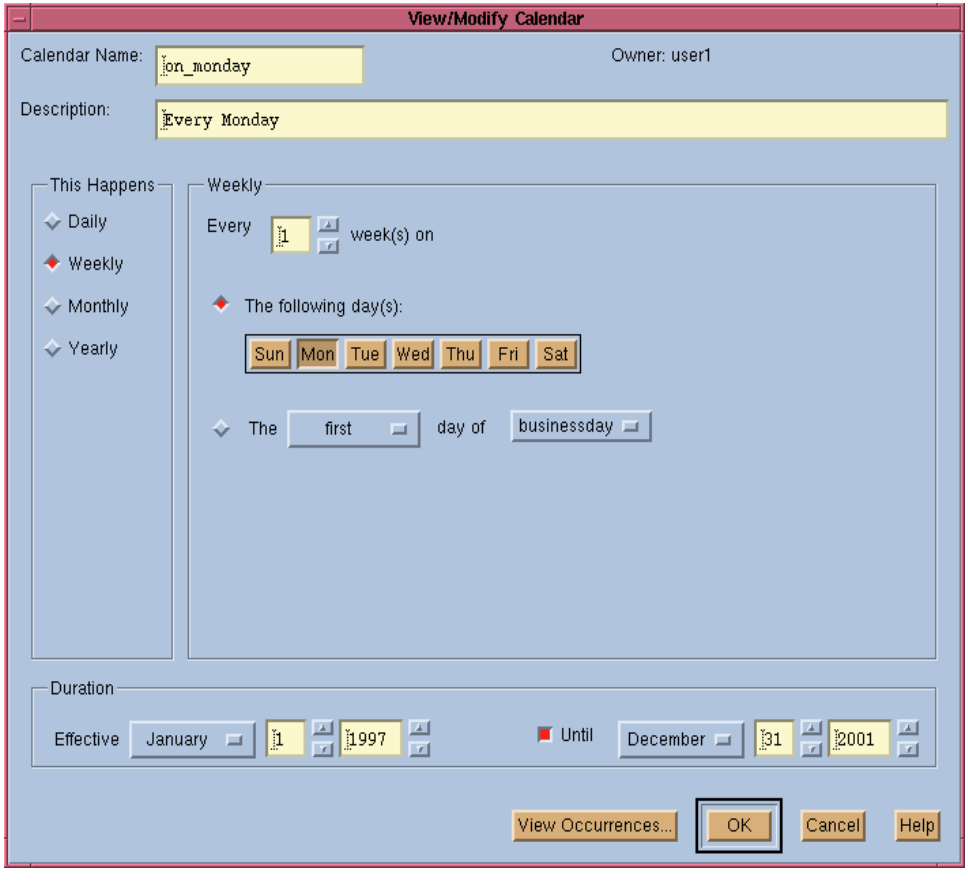
A calendar expression defines the days during which the calendar is active. To prevent a command line interpreter, such as a shell in UNIX, from interpreting any special characters, the calendar expression following the `-t` option of `bcadd` should be placed within quotes. See *'Command Line Interface for Defining Calendars' on page 39* for further details.

Once your calendar is created, it can be used to drive jobs. A calendar can also be modified using `bcmod` command or the `xbcal` GUI.

## Creating a Calendar Using the LSF JobScheduler - Calendar GUI

Although you can create calendars from the command line, it is more convenient to use the LSF JobScheduler - Calendar graphical application to do it. *Figure 2* shows the calendar creation window of the `xbcal` GUI tool.

Figure 2. `xbcal` Calendar Creation Window



# Displaying Calendars

LSF JobScheduler gives you two ways to display calendars—using the `bcal` command from a command-line prompt, or using the `xbcal` graphical tool.

## Displaying Calendars from the Command Line

The `bcal` command displays information about the calendars in the system.

```
% bcal
CALENDAR_NAME  OWNER  STATUS    LAST_CAL_DAY     NEXT_CAL_DAY
on_monday      user1  inactive  Mon Nov 10 1997  Mon Nov 17 1997
workdays       sys    active    Wed Nov 12 1997  Thu Nov 13 1997
```

By default, `bcal` displays all of your calendars, and all system-defined calendars. You can specify a calendar by name.

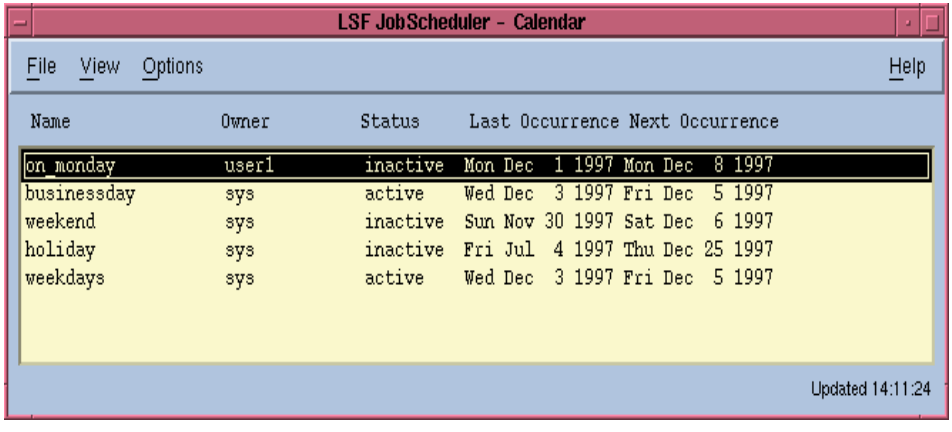
```
% bcal on_monday
CALENDAR_NAME  OWNER  STATUS    LAST_CAL_DAY     NEXT_CAL_DAY
on_monday      user1  inactive  Mon Nov 10 1997  Mon Nov 17 1997
```

See *'Manipulating Calendars Using the Command Line Interface' on page 40* for details of the individual fields of the `bcal` output.

### Displaying Calendars Using the LSF JobScheduler - Calendar GUI

The same information is available using the LSF JobScheduler - Calendar graphical application `xbcal`. In *Figure 3*, `xbcal` has been used to display all the calendars on the system.

Figure 3. `xbcal` Calendar List Window




## Associating Jobs with Calendars

LSF JobScheduler gives you two ways to associate jobs with calendars—using the `bsub` command from a command-line prompt, or using the `xbsub` graphical tool.

### Associating Jobs with Calendars from the Command Line

Once you have created your calendar, you can associate jobs with it. Submit a calendar-driven job using the `bsub` command. The `-T` option defines a calendar-dependent job by specifying the active time within a calendar.

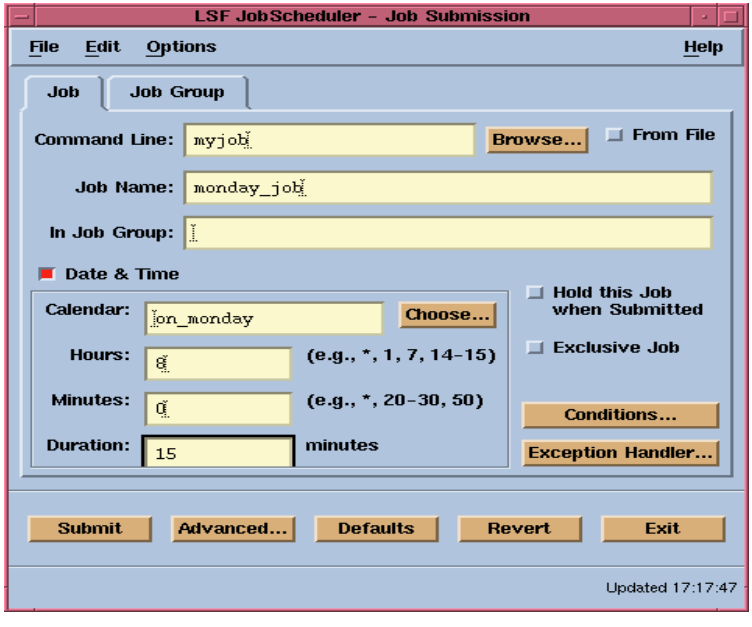
```
% bsub -T "on_monday:01:00" myjob
Job <3456> is submitted to default queue <normal>.
```

This calendar-driven job will be executed each Monday at 1:00 am.

### Associating Jobs with Calendars Using the LSF JobScheduler - Job Submission GUI

You can use the GUI application `xbsub` to associate jobs with your calendar. *Figure 4* shows submitting a job to the calendar ‘on_monday’.

Figure 4. `xbsub` Window




## Deleting Jobs

Use the `bdel` command to remove a job.

```
% bdel 3456
Job <3456> is being deleted
```

This command removes a specific job from the system. If the job is currently running, `bdel` kills the job before removing the job from the system. You can also delete a job using the LSF JobScheduler GUI by selecting on the job from the `xlsjs` main window and clicking “Delete a Job”.

## Associating Jobs with Other Jobs

You can make a job depend on one or more prior jobs. For example, your job may require that the previous job has started processing (but it does not matter if it has completed).

```
% bsub -w “started(first_job)” -J second_job time_card
```

In normal processing, your job probably requires that the prior job finished successfully.

```
% bsub -w “done(pre_process)” -J main_process cheque_run
```

Such operations can also be easily done using JobScheduler’s submission GUI. See *‘Inter-job Dependencies’ on page 83* for information on this graphical tool.

## Associating Jobs with File Events

You may want a job to run after some file event has occurred. LSF JobScheduler will check for the file event and run your job. For example,

```
% bsub -w "file(size(/data/log_file)>40M)" command
```

This tells LSF JobScheduler to run the job when file “`/data/log_file`” has grown beyond 40 MB in size.

```
% bsub -w "file(exist(/data/new_file))" command
```

This tells LSF JobScheduler to run the job if file “new_file” exists.

The LSF JobScheduler GUI xbsub provides you with an easier way to specify file event dependencies. See *'File Event Dependency' on page 86* for all available file event functions, and more examples.

# Tracking Jobs

After you have submitted jobs to LSF JobScheduler, you can check the status of your jobs by running the `bjobs` command.

```
% bjobs
JOBID USER   STAT   QUEUE      FROM_HOST EXEC_HOST JOB_NAME SUBMIT_TIME
1004  user6  RUN    sim        hosta     hosta     portf1   Sep 1 09:23
1235  user6  PEND   priority   hostb               backup   Sep 1 13:55
1234  user6  SSUSP  sim        hostd     hostb     portf2   Sep 1 10:09
1250  user6  PEND   sim        hosta               forecast Sep 1 13:59
```

Running jobs are listed first. Pending jobs are listed in the order in which they will be scheduled. Jobs in high priority queues are listed before those in lower priority queues.

The LSF JobScheduler GUI `xlsjs` provides an easy interface to monitor job status.

# Using LSF JobScheduler GUI Tools

All the operations described above can be executed using the LSF JobScheduler GUI suite *xlsf*.

**`xbsub`** - job submission GUI

**`xbmod`** - this GUI allows you to modify the attributes of already submitted jobs

**`xbcal`** - calendar GUI that allows you to create, remove, and modify calendars

**`xlsjs`** - this GUI allows you to do all of the above including monitoring job execution history, schedules, dependencies, and job groups

**xbalarms -** LSF JobScheduler alarm GUI that allows you to view, acknowledge, and resolve alarms

The online help built into these GUI tools allows you to learn as you work.

# 3. Events and Calendars

LSF JobScheduler manages network-wide events, and uses events to drive the scheduling of jobs. LSF JobScheduler responds to the following types of events:

- **time events -** points of time that can be used to trigger the scheduling of jobs
- **job events -** the starting and completion of other jobs
- **job group events -** job group status conditions which can be used to trigger the execution of jobs that depend on them
- **file events -** changes in the files residing in accessible file systems, such as the arrival (creation) of a specific file
- **user events -** site-specific occurrences—such as a tape mount—defined by the LSF JobScheduler administrators for your system
- **exception events -** conditions which indicate a problem with the processing of a job

File events and user events are handled by an External Event Server (`eeventd`). LSF JobScheduler comes with an `eeventd` daemon that handles file events. A site can modify `eeventd` to monitor any site-specific events and use them to drive jobs.

Since file events and user events are events external to the LSF JobScheduler, they are also referred to as *external events*, whereas time events, job events, job group events, and exception events are referred to as *built-in events*.

Time events are defined by calendars and time expressions, specified when a job is submitted. A calendar is a set of days during which time events occur. A time expression specifies time(s) of the day at which time events occur.

# How Are Events Created?

LSF JobScheduler events are used to trigger jobs. As such, events are defined when jobs that are to be triggered by the events are created. There is a difference between a condition and an event. An event is a condition that has been associated with a job as one of its scheduling conditions. A condition that is not associated with a job is not monitored by LSF JobScheduler, and is not considered an event.

When you create a new job, you can specify one or more conditions that will trigger the job's execution. As a user, you only need to worry about the conditions of your jobs, and LSF JobScheduler will automatically register events that monitor the conditions you specify.

# Event Status

The *status* of an event has three possible values:

- **active -** the event is considered to have occurred. The dependency condition for jobs waiting on the event will evaluate to TRUE.
- **inactive** - the event is considered not to have occurred. The dependency condition for jobs waiting on the event is considered FALSE.
- **invalid** - this applies to external events only. The event is invalid and thus is ignored by `eeventd`. This usually occurs due to a syntax error in the event statement. The dependency condition for jobs waiting on the event will evaluate to FALSE.

At any given instant, the event is either active, inactive, or invalid.

# Built-in Events

*Built-in events* are events inside LSF JobScheduler. These events are monitored by LSF JobScheduler rather than by an `eeventd`.

## Time Events

A calendar together with a time expression defines a sequence of time events. Time events are a useful means of triggering repetitive jobs.

A time event has two attributes: a start time and a duration.

- **start time -** defines when the time event becomes active
- **duration -** specifies how long the time event remains active before it becomes inactive

See *'Calendars and Time Events' on page 31* for detailed information on time events.

## Job Events

Frequently, the status changes of some jobs can trigger the scheduling of other jobs. The change in status of a job is a job event. LSF JobScheduler allows you to submit a job so that the scheduling of this job is dependent on the status of some prior job(s).

The following job event functions are provided to specify an inter-job dependency when you submit a job. For all functions, the parameter `job` is either a jobID or a jobname.

**`started(job)`**
: The condition is TRUE if the specified job has started running or has already finished.

**`done(job)`**
: The condition is TRUE if the specified job has finished successfully in the DONE state. A job is considered to have finished successfully if it terminates with exit code 0 (zero).

**`exit(job)`**
The condition is TRUE if the specified job has terminated abnormally in the EXIT state. A job is considered to have terminated abnormally if it terminates with a non-zero exit code.

**`exit(job,exit_code)`**
The condition is TRUE if the specified job has terminated with the specified `exit_code` value.

**`exit(job,op exit_code)`**
The condition is TRUE if the specified job has terminated with an exit code within the specified range of `exit_code`, where `op` is one of >, >=, <, <=, ==, or !=.

**`ended(job)`**
The condition is TRUE if the specified job has finished.

The jobname can be preceded by a job group specification to indicate a dependency on a job belonging to a particular group. See *'Inter-job Dependencies' on page 83* for examples of using the above functions when submitting a job.

## Job Group Events

A job or a job group can depend on the status of another job group. Using the group dependencies you can set up a sequence of job groups to execute in a particular order. A group itself is not actually executed, but rather the individual jobs under the group. Therefore the successful completion or failure of a group is determined by the state of the jobs in the group.

The following functions are provided to specify job group dependencies:

**`active(group_spec)`**
TRUE if the group is in the ACTIVE state. A group is active if it is ready to schedule jobs.

**`inactive(group_spec)`**
TRUE if the group is in the INACTIVE state, i.e. no job in the group may scheduled to run.

**hold(group_spec)**
TRUE if the group is in the HOLD state.

**numrun(group_spec, op num)**
TRUE if the number of jobs in RUN state satisfy the test.

**numpend(group_spec, op num)**
TRUE if the number of jobs in PEND state satisfy the test.

**numdone(group_spec, op num)**
TRUE if the number of jobs in DONE state satisfy the test.

**numexit(group_spec, op num)**
TRUE if the number of jobs in EXIT state satisfy the test.

**numended(group_spec, op num)**
TRUE if the total number of jobs in the DONE or EXIT state satisfy the test.

**numstarted(group_spec, op num)**
TRUE if the total number of jobs in the RUN, USUSP or SSUSP state satisfies the test.

## Job Exception Events

Exceptions are conditions that arise during the life of a job that may require notification or corrective action. An exception can trigger either a corrective job (by means of an exception event), or an exception handler (such as an alarm).

Since every job can potentially display unique behaviour, the definition of an exception is entirely up to the user who creates the job, i.e. a condition that is considered to be an error for one job may not be considered an error for another job. LSF JobScheduler allows each user to have his/her own definition of exceptions for each of his/her jobs.

When you create a job, you can specify what you would consider to be an exception for that job. You can then specify exception handlers to take care of the exception conditions. When an exception occurs, the associated exception handler will be automatically invoked to take action. Exception handling is discussed in more detail in *Section 7, 'Exception Handling and Alarms', beginning on page 123.*

This section focuses on exception events that can be created as a result of exception handling. LSF JobScheduler provides many exception handlers to recover a job when errors occur. These handlers allow you to specify how you want to handle the job when certain exceptions happen. For example, you can re-run the job, terminate the job, trigger an alarm, or trigger an external exception handler job. Exception events are used to trigger external exception handlers.

As with all other events, an exception event is created when a job that responds to the event is created. An exception event has a name defined by the user who creates the job. The following is an example of how an exception event can be created:

```
% bsub -w "exception(too_long)" re-init
```

This command submits a job `re-init` that runs when exception event `too_long` occurs. LSF JobScheduler then registers an exception event named `too_long`. Note that the full name of the event is `too_long@user` where `user` is the owner of the job. This allows different users to use the same exception name without causing conflicts.

For the registered exception event to become active, a source for the exception must also be defined. This is done through the special exception event handler function setexcept():

```
% bsub -X "overrun(60)::setexcept(too_long)" simulation
```

Here, `overrun()` is an exception function and `setexcept()` is an exception handler that sets the exception event `too_long` to active when the exception condition `overrun(60)` becomes TRUE. The exception condition `overrun(60)` becomes TRUE when the job runs for more than 60 minutes.

In the above example, the `re-init` job creates the event and serves as the external event handler, and the simulation job sets the event to active when the exception happens. It is possible to have more than one job handling the same exception event. In this case, the first job creates the event and other jobs would refer to the event. The event is removed from the system when all exception handling jobs for the event are removed from LSF JobScheduler.

For a list of all valid exception conditions and exception handlers in LSF JobScheduler see *Section 7, 'Exception Handling and Alarms', beginning on page 123*.

# External Events

External events allow your jobs to be triggered by conditions external to LSF JobScheduler. This provides a lot of flexibility for you to integrate your site-specific conditions with the scheduling of your production jobs.

Typical examples of external events include the arrival of a file, the mount of a device, and the detection of an exception situation in the system.

External events are detected by the External Event Daemon (`eeventd`), which resides on the same server host as the master scheduler daemon (`mbatchd`). The `eeventd` communicates with `mbatchd` using a well-defined protocol. LSF JobScheduler comes with an `eeventd` that detects file events. A site can easily modify the `eeventd` to integrate other events by adding event processing functions.

## File Events

A file event condition is specified as:

```
file(file_condition_expression)
```

where the keyword `file()` tells the master scheduler (`mbatchd`) that this is a file condition so that the parameter *file_condition_expression* should be passed to the `eeventd` for processing. The *file_condition_expression* is a logical expression in terms of the following four file status functions:

**`arrival(file_loc)`**

This function evaluates to TRUE when the file specified by *file_loc* arrives. The arrival of a file refers to the transition from non-existence to existence of the file.

**`exist(file_loc)`**

This function evaluates to TRUE if the file specified by *file_loc* exists. Note that this function is different from `arrival()` in that a transition from a non-existence to existence is not needed. As long as the file exists, the function always evaluates to TRUE.

**size(file_loc)**

This function evaluates to the size of the file specified by *file_loc* in bytes. If the file does not exist, this function evaluates to 0.

**age(file_loc)**

This function evaluates to the age of the file specified by *file_loc* since the last modification in minutes. If the file does not exist, this function evaluates to 0.

The `file_loc` in the above functions takes the following form:

```
[hostname:]absolute_directory/filename
```

Here, `hostname` is the name of the host on which the file can be accessed. Note that this host does not have to be the same host on which the job executes. If `hostname` is not specified, then the Event Server assumes that the file is accessible from any host.

**Note**

*You must specify the absolute path name of the file being evaluated in the above expressions.*

An example of a file event condition using the above file event functions is:

```
file(exist(/tmp/core) && size(/tmp/core)>10M)
```

A file event is automatically created when a user submits a job with a file event dependency condition. The event is automatically removed when there are no dependent jobs.

See *'File Event Dependency' on page 86* for examples of how to use file event conditions when submitting a job.

## User Events

LSF JobScheduler provides an open mechanism for sites to implement site specific events by adding more event processing functions into the External Event Daemon (`eeventd`). A user event condition is specified as:

```
event(event_spec)
```

where the keyword `event()` tells the master scheduler (`mbatchd`) that this is a user event so that the parameter *event_spec* should be passed to `eeventd` for processing. The site is responsible for writing the function that parses and processes the *event_spec*. See *"External Event Management"* in the *LSF JobScheduler Administrator's Guide* for details of how to modify `eeventd`.

A user event can be used to detect arbitrary site specific environmental status that can trigger production jobs. For example, a '`diskisfull`' event could be designed to detect the fact that a critical file system is full, and therefore, an exception handling job should be triggered to correct the situation. In this case, the user event condition might be specified as:

```
event(diskisfull)
```

This will create an event '`diskisfull`' which is monitored by the `diskisfull` event processing function in the `eeventd`.

See *'User Event Dependency' on page 90* for examples of associating user events with job submissions.

# Viewing Events

You can view all events in LSF JobScheduler. To view prior job events and job group events, use the `bjobs` command. To know the status of a time event, simply view the time event definition of the job by looking at the detailed job definition information, and by looking at the calendar status using a calendar tool, such as `bcal` or the `xbcal` GUI.

To view exception events and external events, use the `bevents` command.

### Examples

```
% bevents

EVENT            OWNER  STATUS    SOURCE  ATTRIBUTE  LAST_UPDATE
size(/tmp/file)  user1  inactive  file    -          Nov 19 18:09:50 1997
too_long@user1   user1  inactive  except  overrun    Nov 19 18:27:01 1997
```

To view all details of an event, use long format:

```
% bevents -l

EVENT: size(/tmp/file)>45M

OWNER  STATUS    SOURCE NUM_OF_DEPENDENTS LAST_UPDATE
user1  inactive  file   1                 Nov 19 18:09:50 1997

LAST_DISPATCHED_JOBID   LAST_DISPATCH_TIME
-                       -

ATTRIBUTE:      -

EVENT: too_long@user1
OWNER  STATUS    SOURCE     NUM_OF_DEPENDENTS  LAST_UPDATE
mike   inactive  exception  1                  Nov 19 18:27:01 1997

LAST_DISPATCHED_JOBID   LAST_DISPATCH_TIME
-                       -

ATTRIBUTE:  "overrun Job[105] User[user1] Queue[priority]"
```

The long format shows you the complete event information. In the above example, the first event is a file event that watches the size of the file, and has one job—which has never run—depending on it. If a job that depends on the event had been triggered by the event, the LAST_DISPATCHED_JOBID and LAST_DISPATCH_TIME would indicate the job ID and the time the job was dispatched.

The second event in the above example is an exception event and has one job—which has never run—depending on it. The ATTRIBUTE parameter is event type-dependent information. This information is passed to jobs that are triggered by the event via the

environment variable LSB_EVENT_ATTRIB. For file events, ATTRIBUTE is empty. For exception events, ATTRIBUTE gives the exception function, job ID of the job that caused the exception, login name of the user who owns the event, and name of the queue to which the job belongs. If you want to define an error recovery job in response to an exception event, for example, you can use the ATTRIBUTE to find the context under which the error occurs.

# Calendars and Time Events

## What is a Calendar?

A calendar is a set of days defined using one or more calendar rules. A calendar, together with a time expression, defines a sequence of time events that can trigger the execution of repetitive jobs. Calendars are defined and manipulated independently of jobs. This allows multiple jobs to share the same calendar. There are three types of calendars you can use to submit your job:

- built-in calendars
- preconfigured system calendars
- user-defined calendars

A calendar has a name, an owner (user), and a description. The name of the calendar and its owner are assigned when it is created, and are case-insensitive. The status of a calendar is determined by whether the current day is one of the days specified in the calendar definition. A calendar is active if the current day is in the calendar's list of days, otherwise it is inactive.

## Built-in Calendars and Reserved Names for Calendars

Built-in calendars are provided in LSF JobScheduler to reflect the most commonly used set of days. You can use these calendars directly without having to define them first.

The available built-in calendars supported in LSF JobScheduler include:

**Sun, Mon, Tue, Wed, Thu, Fri, Sat**

These are the days of the week. For example, the calendar "Sun" means every Sunday.

**Day**

This calendar refers to every day.

Since built-in calendars have obvious meaning in daily life, you cannot view the status of built-in calendars.

LSF JobScheduler reserves the names of all built-in calendars. You cannot create a user calendar that conflicts in name with one of the built-in calendars.

In addition to built-in calendars, LSF JobScheduler also reserves the following names. These names are not calendar names but they are reserved as building blocks of calendar definition. See *'Calendar Expressions and the Command Line Interface' on page 37* for details of the use of these keywords in the definition of calendars.

Listed below are reserved keywords that are not built-in calendar names by themselves.

**Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec**

These are months of the year and are reserved as building blocks of calendar definitions. See *'Calendar Expressions and the Command Line Interface' on page 37* for detailed usage information.

**Week**

This keyword is reserved for use in calendar definitions to specify a period of a week.

**Month**

This keyword is reserved for use in calendar definition to specify a period of a month.

**Quarter**

This keyword is reserved for use in calendar definition to specify a period of a quarter.

**YY**

This keyword is reserved for use in calendar definition to specify a set of years.

## System Calendars

System calendars are read-only calendars defined in the LSF JobScheduler configuration by the LSF administrator. System calendars are owned by the virtual user “sys”, and can be viewed by everybody. You cannot modify or delete system calendars.

**Note**

*The user account “sys” does not need to exist in the system.*

System calendars can be used as normal calendars. When a system calendar is defined, its name becomes a reserved calendar name in the cluster. When the LSF JobScheduler daemons start up, the system calendars are defined in the cluster.

## Using the LSF JobScheduler - Calendar GUI

To create a calendar you can use either command line or GUI tools. The GUI tool `xbcal` lets you create, view, and manipulate calendars.

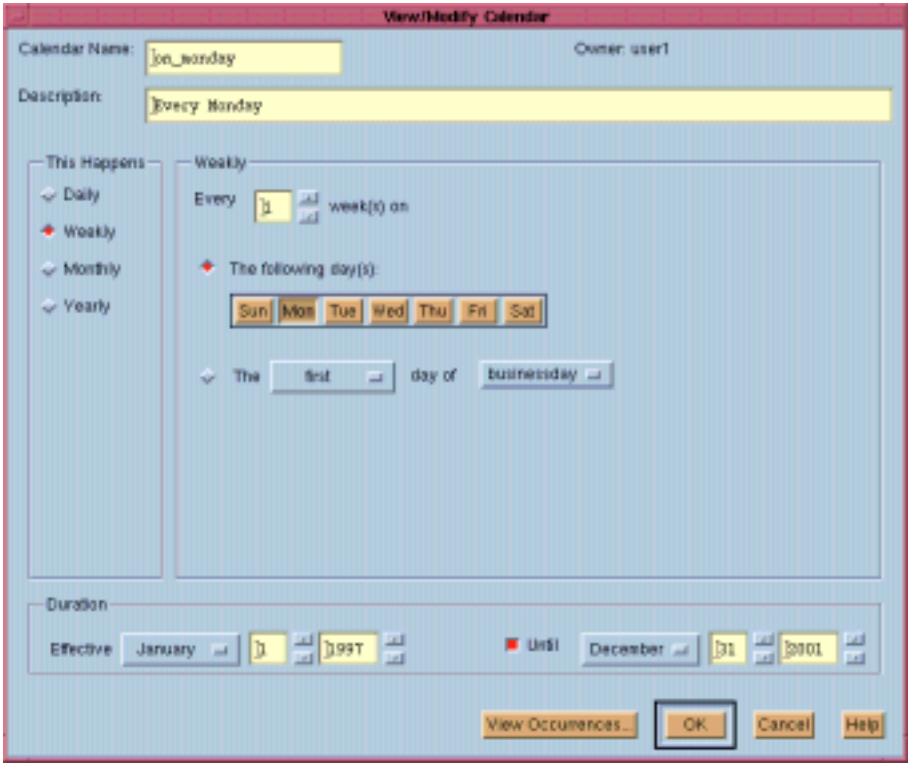
### Creating Calendars

`xbcal` supports three ways of defining calendars.

- clicking on dates
- specifying recurrence patterns
- combining existing calendars

*Figure 5* shows the `xbcal` screen that is displayed by choosing **File | New Calendar| by Specifying Recurrence Pattern**.

Figure 5. Adding the "on_monday" calendar

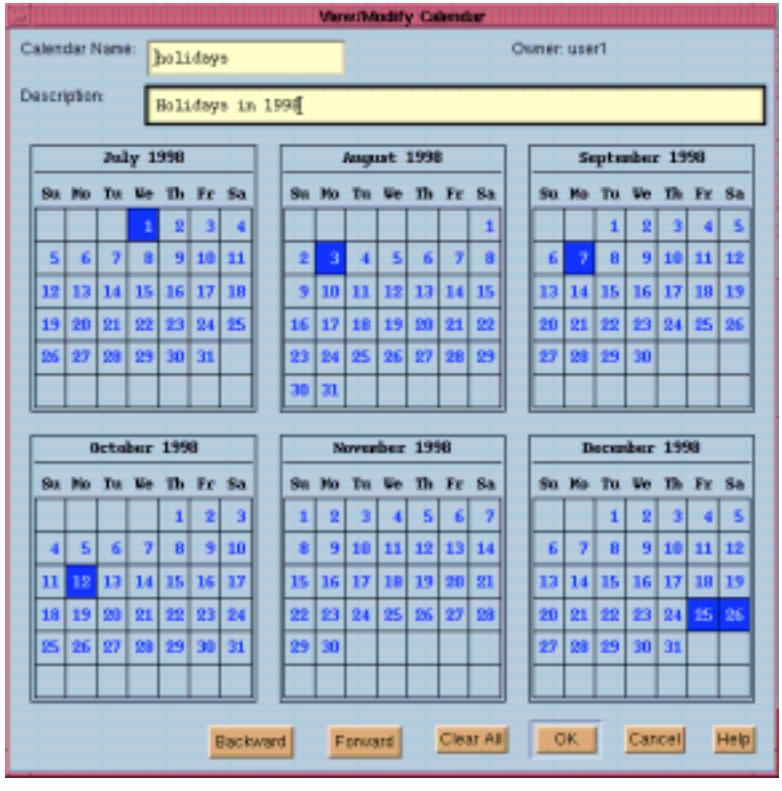



If you wish to define a set of days with a regular recurrence pattern, you can use the window shown in *Figure 5* to create the calendar.

When the set of days you wish to define does not have any regular pattern, use the "by Clicking on Dates" window. A natural calendar is displayed and you can click on

particular days that define the calendar. *Figure 6* is the GUI interface for defining a calendar by selecting specific days.

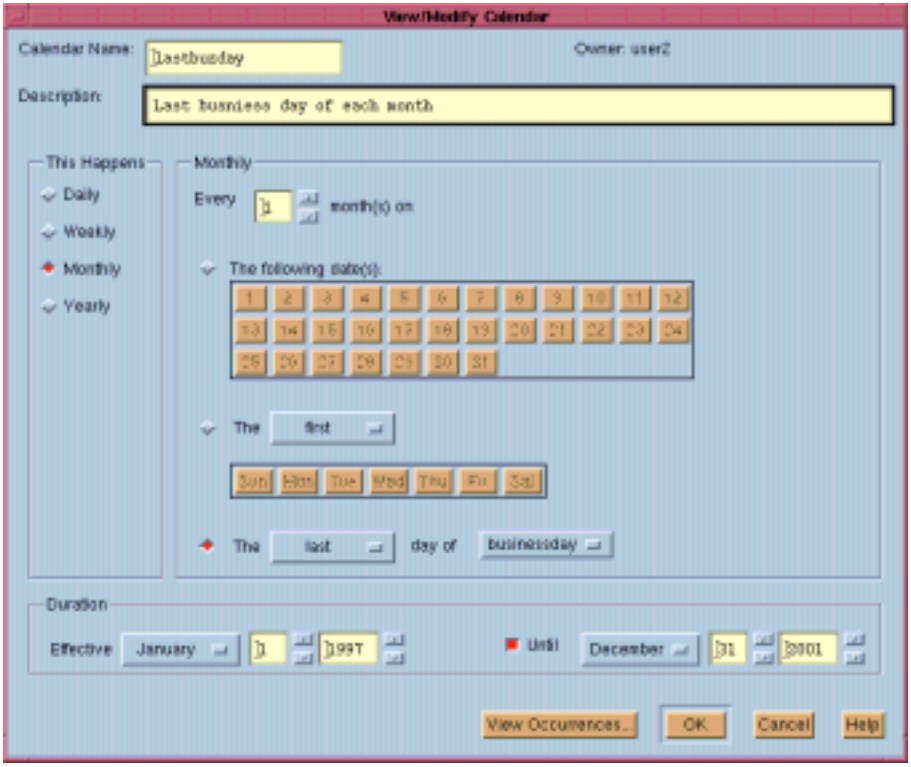
Figure 6. Defining a Calendar by Selecting Specific Days




LSF JobScheduler also provides the flexibility to create calendars on top of existing calendars. For example, if you already have a calendar named “businessdays”, you can define a new calendar called “lastbuzday” using the existing calendar businessdays.

*Figure 7* shows the window for creating the “lastbuzday” calendar out of the existing “businessdays” calendar.

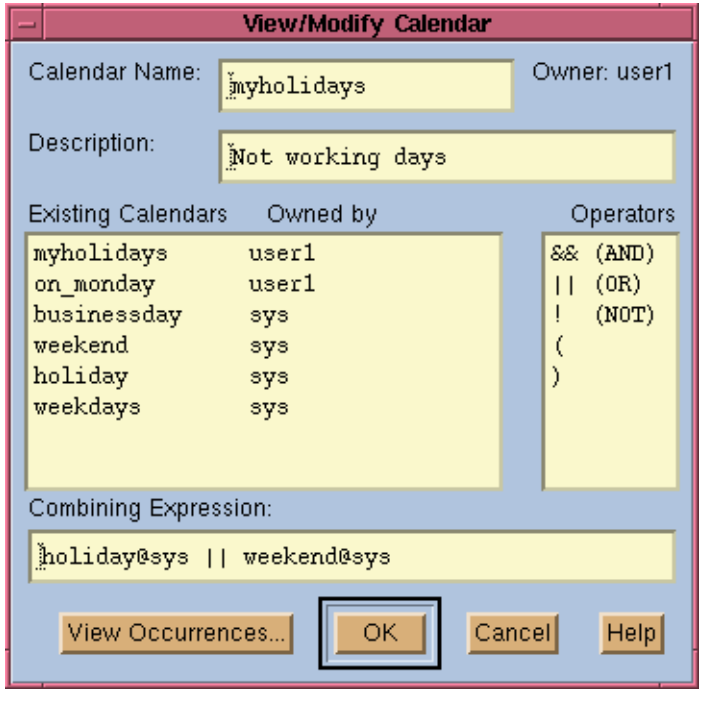
Figure 7. Creating the “lastbuzday” Calendar




The windows shown in *Figure 7* and *Figure 8* allow you to create a new calendar by combining multiple calendars using logical expressions. The "AND" operator selects days that are common to the two calendars, whereas the "OR" operator merges the days of both calendars. The "NOT" operator selects all days that are not part of a

calendar. The "View Occurrences" button creates a popup window that displays the actual days of the newly combined calendar.

Figure 8. Combining Existing Calendars




## Calendar Expressions and the Command Line Interface

Calendars can also be defined using the command line interface provided by LSF JobScheduler. In order to use commands to manipulate calendars, you first need to understand the concept of *calendar expressions.* A calendar expression in LSF JobScheduler is a powerful calendar definition language that provides flexible ways to define arbitrary sets of days.

### Simple Calendar Expressions

A simple calendar expression takes one of the following formats:

- **YEAR:MONTH:DAY**
- **YEAR:WEEK:DAY**

The "YEAR" field defines the set of years during which the set of days will be chosen. Valid values for the "YEAR" field can be any one year, or a list of years separated by commas, such as "1997, 1998". You can also use the keyword "YY" to specify a recurring list of years in the following format:

```
YY(start_year, end_year, step)
```

Here, `start_year`, `end_year`, and `step` are integers.

The "MONTH" field specifies the set of months within the years defined by the "YEAR" field. The format of the "MONTH" field can be one or more integers in the range of 1 to 12, separated by commas, such as "1, 3, 5", or one or more of the keywords from “Jan” to “Dec”. You can also use the keyword "Month" to specify a recurring list of months in the following format:

```
Month(start_month, end_month, step)
```

Here, `start_month`, `end_month`, and `step` are integers.

The "WEEK" field defines the set of weeks within the years defined by the "YEAR" field. The format of the "WEEK" field is:

```
Week(start_week, end_week, step)
```

Here, `start_week`, `end_week`, and `step` are integers.

The "DAY" field defines the set of days within the specified months of the year, or weeks of the year. You can specify multiple days separated by commas. Each day can be specified by an integer between 1 and 31 for the days of the month, or between “Sun” and “Sat” for the days of the week. To specify recurring days, you can also use:

```
Day(start_day, end_day, step)
```

Here, `start_day`, `end_day`, and `step` are all integers.

A special character "*" can be used in any field above to mean "every year", "every month", or "every day".

For each of the reserved keywords described in *'Built-in Calendars and Reserved Names for Calendars' on page 31*, you can also use sub-indices to select a day, week, month, quarter or year in no particular order, or to select a particular day, week, month, quarter, or year relative to the start of the range.

For instance, "Mon(-1)" refers to last Monday, "Day(-2)" refers to the second last day, and "Week(3)" refers to the third week.

For examples of simple calendar expressions, see *'Command Line Interface for Defining Calendars' on page 39*.

### Command Line Interface for Defining Calendars

Although it is easier to define calendars using the `xbcal` GUI interface, LSF JobScheduler also provides a command line interface for calendar manipulations. Calendars can be created using the `bcadd` command. Below are some examples of creating calendars using `bcadd`:

```
% bcadd -d "Back up days on Friday" -t "*:*:Fri" backup_days
```

This creates a calendar named "backup_days" that includes every Friday. The `-d` option allows you to give a description of your calendar.

```
% bcadd -d "bi-weekly pay days on Friday" -t "*:Week(1,*,2):Fri" pay_days
```

This creates a calendar that is active every two weeks, on Fridays, starting from the beginning of each year.

```
% bcadd -d "Last Friday of every July" -t "*:Jul:Fri(-1)" report_days
```

This creates a calendar that is active on the last Friday of July of every year.

```
% bcadd -d "Quarterly synchronization days" -t "*:quarter:day(1)" quarterly
```

This creates a calendar that is active on the first day of each quarter.

### Complex Calendars

Simple calendar expressions give you a way to define a calendar that is straightforward. In some cases, a calendar can be too sophisticated to be defined in a single calendar. For example, suppose you want to define a calendar that is active on all US holidays and Canadian holidays, but not if it is a Wednesday. It would be difficult to define this using simple calendar expressions.

A combined calendar expression introduces logical operations into calendar definition and provides a structured way to construct complex calendars out of simple calendars.

A combined calendar expression consists of one or more simple calendar expressions and one of more of the logical operators "&&" (AND), "||" (OR), and "!" (NOT). Multiple levels of logical expressions can be constructed by using "(" and ")" to group expressions in desired order.

The "&&" operator selects days that exist in both calendars, while the "||" operator merges all days in both calendars together. The "!" operator specifies days that are not contained in any calendar.

For example, to construct the calendar mentioned above, you would first define a us_holidays calendar and a canadian_holidays calendar using simple calendar expression, then create a complex calendar using:

```
% bcadd -t "(canadian_holidays || us_holidays) && ! Wed"  na_holidays
```

**Note**

*Since "Wed" is a built-in calendar, you do not need to define it beforehand.*

## Manipulating Calendars Using the Command Line Interface

Although you can do all calendar-related operations through the GUI tools, LSF JobScheduler also includes the command line tools necessary for you to manipulate your calendars.

Calendars can be displayed using the `bcal` command:

```
% bcal

CALENDAR_NAME  OWNER  STATUS    LAST_CAL_DAY      NEXT_CAL_DAY
businessdays   sys    active    Thu Nov 20 1997   Mon Nov 24 1997
weekdays       sys    active    Thu Nov 20 1997   Mon Nov 24 1997
holidays       sys    inactive  Fri Jul  4 1997   Thu Dec 25 1997
on_monday      user1  inactive  Mon Nov 17 1997   Mon Nov 24 1997
```

By default, `bcal` shows all system calendars and the user's own calendars. You can view other users' calendars by using the `-u` option of the `bcal` command.

To know more details about each calendar, you can use the `-l` option of the `bcal` command:

```
% bcal -l quarterly
CALENDAR:  quarterly
-- First day of each quarter

OWNER  STATUS     CREATION_TIME              LAST_MODIFY_TIME
user1  inactive   Fri Nov 14 17:50:01 1997    -

CAL_EXPRESSION:  *:quarter:day(1)
LAST_CAL_DAY:     <Wed Oct 1 1997>
NEXT_CAL_DAY:     <Thu Jan 1 1998>
```

After a calendar is created, you can modify it using the `bcmod` command:

```
% bcmod -d "New description: quarterly monday" -t "*:quarter:mon(1)" quarterly
```

This overwrites the previous definition of calendar "quarterly". You can only modify your own calendars.

To delete a calendar, use the `bcdel` command:

```
% bcdel quarterly
```

## Time Events

*Time events* in LSF JobScheduler are durations of time that are used as jobs' execution conditions. A time event is defined when a job is created with a time event dependency condition. The time event can be specified in the "Date & Time" area of the `xbsub` GUI, or via the `-T` option of the `bsub` command.

A time event has a start date and time at which the event becomes active, and a duration in which the event remains active. The start date can be represented by a calendar, and the start time and duration must be specified for each job.

When a job is submitted with a time event, LSF JobScheduler monitors the time event status. Once current time falls within the start time and duration, the time event becomes active and triggers the job execution.

Figure 9. Defining a Time Event

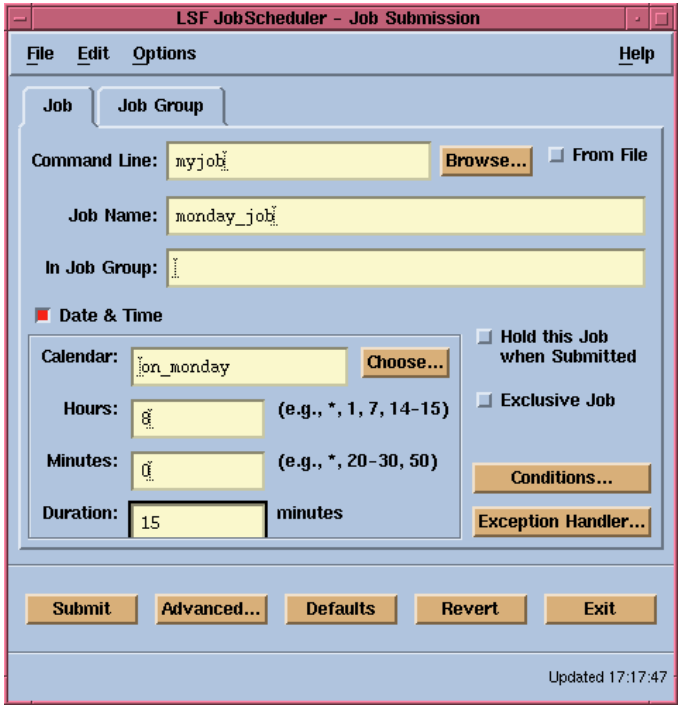


*Figure 9* is an example of how a time event can be defined when submitting a job using the `xbsub` GUI.

Note that multiple hours and minutes can be put in the time area to create a time event that repeats multiple times per day. In particular, "*" can be put in the "Hours" or "Minutes" areas to refer to "every hour" or " every minute".

### Time Expressions and the Command Line Interface

To define a time event for a job from the command-line interface, a time expression can be specified. A time expression has the following format:

```
[calendar_name[@user_name]:]hours:minutes[%duration]
```

Here, `@user_name` and `%duration` are optional. By default, a user will be using his/her own calendars and system calendars. If you intend to use another user's calendar, you must use "`calendar_name@user_name`" to explicitly specify the owner of the calendar.

If a duration is not specified, LSF JobScheduler assumes a default of one minute.

If a calendar is not explicitly specified, LSF JobScheduler assumes the built-in calendar, "Day", as the calendar. The built-in calendar "Day" means every day. See *'Built-in Calendars and Reserved Names for Calendars' on page 31* for detailed information on built-in calendars.

With time expressions, you can submit jobs associated with time events from the command line. For example:

```
% bsub -T "on_monday:2:00%60" backup_job
```

This creates a job "backup_job" that will run every Monday between 2:00AM and 3:00AM. The duration is 60 minutes indicating that the event will be active at 2:00AM and remain active until 3:00AM.

If the job is unable to start by 3:00AM, it is considered to have missed its schedule and will not be scheduled until next time the event becomes active again. You can define exception handlers to handle such situations.

See *Section 7, 'Exception Handling and Alarms', beginning on page 123* for details on exception handling.

# 4. Resources

This section describes the system resources monitored by LSF JobScheduler, and the use of resource specifications. Specific topics covered in this section are:

- resources
- load indices
- resource requirement specifications
- how to use task lists to set resource requirements for applications

## Introduction to Resources

A computer network may be thought of as a collection of resources used to execute programs. Different applications often require different resources. For example, a number crunching program may take a lot of CPU power, but a large spreadsheet may need a lot of memory to run well. Some applications may run only on machines of a specific type, and not on others.

In LSF JobScheduler, resources are handled by naming them and tracking information relevant to them. LSF JobScheduler does its scheduling according to applications' resource requirements and the resources available on individual hosts. Resource names are case sensitive, and can be up to 29 characters in length (excluding some characters reserved as operators in resource requirement strings).

LSF JobScheduler classifies resources in different ways.

## Classification by Availability

**general resources**
These are resources that are available on all hosts, e.g. all the load indices, number of processors on a host, total swap space, host status.

**special resources**
These are resources that are only associated with some hosts, e.g. FileServer, solaris, compServer.

## Classification by the Way Values Change

**dynamic resources**
These are resources that change their values dynamically, e.g. all the load indices, host status.

**static resources**
These are resources that do not change their values dynamically, e.g. all resources except load indices and host status are static resources.

## Classification by Types of Values

**numerical resources**
These are resources that take numerical values, e.g. all the load indices, number of processors on a host, host CPU factor.

**string resources**
These are resources that take string values, e.g. host type, host model, host status.

**boolean resources**
These are resources that denote the availability of specific features, e.g. hspice, FileServer, SYSV, aix.

## Classification by Definition

**configured resources**
These are resources defined by user sites, e.g. all resources defined in the LSF configuration files.

**built-in resources**

These are resources that are always defined by LSF, e.g. load indices, number of CPUs, total swap space.

### Classification by Location

**host-based resources**

These are resources that are not shared among hosts, but are tied to individual hosts, e.g. CPU, memory, swap space. An application must run on a particular host to access such resources.

**shared resources**

These are resources that are not associated with individual hosts in the same way, but are "owned" by the entire cluster, or a subset of hosts within the cluster, e.g. floating licenses, shared file systems. An application can access such a resource from any host which is configured to share it, but doing so affects its value as seen by other hosts.

You can list the resources available in your cluster using the `lsinfo` command.

## Load Indices

Load indices measure the availability of dynamic, non-shared resources on hosts in the LSF cluster. Load indices built into the LIM are updated at fixed time intervals. *External load indices* are updated when new values are received from the external load collection program, ELIM, configured by the LSF administrator. Load indices are numeric in value.

*Table 1* summarizes the load indices collected by the LIM.

Table 1. Load Indices

| Index | Measures | Units | Direction | Averaged over | Update Interval |
|---|---|---|---|---|---|
| `r15s` | run queue length | processes | increasing | 15 seconds | 15 seconds |
| `r1m` | run queue length | processes | increasing | 1 minute | 15 seconds |
| `r15m` | run queue length | processes | increasing | 15 minutes | 15 seconds |
| `ut` | CPU utilization | (per cent) | increasing | 1 minute | 15 seconds |
| `pg` | paging activity | pages in + pages out per second | increasing | 1 minute | 15 seconds |
| `ls` | logins | users | increasing | N / A | 30 seconds |
| `it` | idle time | minutes | decreasing | N / A | 30 seconds |
| `swp` | available swap space | megabytes | decreasing | N / A | 15 seconds |
| `mem` | available memory | megabytes | decreasing | N / A | 15 seconds |
| `tmp` | available space in temporary file system | megabytes | decreasing | N / A | 120 seconds |
| `io` | disk I / O (shown by `lsload -l`) | kilobytes per second | increasing | 1 minute | 15 seconds |
| *name* | external load index configured by LSF administrator | | | | site defined |

Load indices can be viewed using the `lsload` command:

```
% lsload
HOST_NAME  status  r15s  r1m  r15m  ut   pg   ls   it  tmp   swp   mem
hosta      ok      0.0   0.0  0.1   1%   0.0  1    224 43M   67M   3M
hostb      ok      0.0   0.0  0.0   3%   0.0  3    0   38M   40M   7M
hostc      busy   *6.2   6.9  9.5   85%  1.1  30   0   5M    400M  385M
hostd      busy    0.1   0.1  0.3   7%   *17  6    0   9M    23M   28M
hoste      unavail
```

The `r15s`, `r1m` and `r15m` load indices are the 15-second, 1-minute and 15-minute average CPU run queue lengths. This is the average number of processes ready to use the CPU during the given interval.

On multiprocessor systems more than one process can execute at a time. LSF scales the run queue value on multiprocessor systems to make the CPU load of uniprocessors and multiprocessors comparable. The scaled value is called the *effective run queue length*. The `-E` option of `lsload` shows the effective run queue length.

LSF also adjusts the CPU run queue based on the relative speeds of the processors (the CPU factor). The *normalized run queue length* is adjusted for both number of processors and CPU speed. The host with the lowest normalized run queue length will run a CPU intensive job the fastest. The `-N` option shows the normalized CPU run queue lengths.

## Static Resources

Static resources represent host information that does not change over time, such as the maximum RAM available to user processes, or the number of processors in a machine.

Most static resources are determined by the LIM at startup time. *Table 2* lists the static resources reported by LSF.

Table 2. Static Resources

| Index | Measures | Units | Determined by |
|---|---|---|---|
| `type` | host type | string | configuration |
| `model` | host model | string | configuration |
| `hname` | host name | string | configuration |
| `cpuf` | CPU factor | relative | configuration |
| `server` | host can run jobs | boolean | configuration |
| `rexpri` | execution priority (UNIX only) | `nice(2)` argument | configuration |
| `ncpus` | number of processors | processors | LIM |
| `ndisks` | number of local disks | disks | LIM |
| `maxmem` | maximum RAM memory available to users | megabytes | LIM |
| `maxswp` | maximum available swap space | megabytes | LIM |
| `maxtmp` | maximum available space in temporary file system | megabytes | LIM |

The `type` and `model` static resources are strings specifying the host type and model.

The CPU factor is the speed of the host's CPU relative to other hosts in the cluster. If one processor is twice the speed of another, its CPU factor should be twice as large. The CPU factors are defined by the LSF administrator. For multiprocessor hosts the CPU factor is the speed of a single processor; LSF automatically scales the host CPU load to account for additional processors.

The `server` static resource is boolean; its value is 1 if the host is configured to run jobs and 0 if the host is a client.

Static resources can be used to select appropriate hosts for particular jobs based on binary architecture, relative CPU speed, and system configuration.

To view the static resources are available in your cluster, run the `lshosts` command.

### Shared Resources

A *shared resource* is a resource that is not tied to a specific host, but is associated with the entire cluster, or a specific subset of hosts within the cluster. Examples of shared resources include:

- floating licenses for software packages
- disk space on a file server which is mounted by several machines
- the physical network connecting the hosts

An application may use a shared resource by running on any host from which that resource is accessible. For example, in a cluster in which each host has a local disk but can also access a disk on a file server, the disk on the file server is a shared resource, and the local disk is a host-based resource. There will be one value for the entire cluster which measures the utilization of the shared resource, but each host-based resource is measured separately.

A shared resource may be configured to be dynamic or static. In the above example, the total space on the shared disk may be static while the amount of space currently free is dynamic. A site may also configure the shared resource to report numeric, string or boolean values.

All shared resources must be configured by the LSF administrator. Run the `lshosts -s` command to view the static shared resources currently configured in your cluster. Run the `lsload -s` command to view the dynamic shared resources configured in your cluster.

## Resource Requirement Strings

A resource requirement string describes the resources a job needs. LSF JobScheduler uses resource requirements to select hosts for remote execution and job execution.

A resource requirement string is divided into three sections:

- a *selection* section: specifies the criteria for selecting hosts from the system
- an *ordering* section: indicates how the hosts that meet the selection criteria should be sorted
- a *resource usage* section: specifies the expected resource consumption of the task

The syntax of a resource requirement expression is:

```
select[selectstring] order[orderstring] rusage[usagestring]
```

**Note**

*The square brackets are an essential part of the resource requirement expression.*

The section names are `select`, `order`, and `rusage`. The syntax for each of `selectstring`, `orderstring` and `usagestring` is defined below.

If no section name is given, then the entire string is treated as a selection string. The `select` keyword may be omitted if the selection string is the first string in the resource requirement.

## Selection String

The selection string specifies the characteristics a host must have to match the resource requirement. It is a logical expression built from a set of resource names. The `lsinfo` command lists all the resource names and their descriptions. The resource names `swap`, `idle`, `login`, and `cpu` are accepted as aliases for `swp`, `it`, `ls`, and `r1m` respectively.

The selection string can combine resource names with logical and arithmetic operators. Non-zero arithmetic values are treated as logical TRUE, and zero as logical FALSE. Boolean resources (for example, `server` to denote LSF server hosts) have a value of one if they are defined for a host, and zero otherwise.

*Table 3* below shows the operators that can be used in selection strings. The operators are listed in order of decreasing precedence.

Table 3. Operators in Resource Requirements

| Syntax | Meaning |
|---|---|
| `-a`<br>`!a` | Negative of `a`<br>Logical not: `1` if `a==0`, `0` otherwise |
| `a * b`<br>`a / b` | Multiply `a` and `b`<br>Divide `a` by `b` |
| `a + b`<br>`a - b` | Add `a` and `b`<br>Subtract `b` from `a` |
| `a > b`<br>`a < b`<br>`a >= b`<br>`a <= b` | `1` if `a` is greater than `b`, `0` otherwise<br>`1` if `a` is less than `b`, `0` otherwise<br>`1` if `a` is greater than or equal to `b`, `0` otherwise<br>`1` if `a` is less than or equal to `b`, `0` otherwise |
| `a == b`<br>`a != b` | `1` if `a` is equal to `b`, `0` otherwise<br>`1` if `a` is not equal to `b`, `0` otherwise |
| `a && b` | Logical AND: `1` if both `a` and `b` are non-zero, `0` otherwise |
| `a \|\| b` | Logical OR: `1` if either `a` or `b` is non-zero, `0` otherwise |

The selection string is evaluated for each host. If the result is non-zero, then that host is selected. For example:

```
select[(swp > 50 && type == MIPS) || (swp > 35 && type == ALPHA)]

select[((2*r15s + 3*r1m + r15m) / 6 < 1.0) && !fs && (cpuf > 4.0)]
```

For the string resources `type` and `model`, the special value `any` selects any value and `local` selects the same value as that of the local host. For example, `type==local` selects hosts of the same type as the host submitting the job. If a job can run on any type of host, include `type==any` in the resource requirements. If no `type` is specified, the default is `type==local` unless a model or boolean resource is specified, in which case it is `type==any`.

## Order String

The order string allows the selected hosts to be sorted according to the values of resources. The syntax of the order string is:

```
[-]res[:[-]res]...
```

Each `res` must be a dynamic load index; that is, one of the indices `r15s`, `r1m`, `r15m`, `ut`, `pg`, `io`, `ls`, `it`, `tmp`, `swp`, `mem`, or an external load index defined by the LSF administrator. For example, `swp:r1m:tmp:r15s` is a valid order string.

**Note**

*The values of* `r15s`, `r1m`, *and* `r15m` *used for sorting are the normalized load indices returned by* `lsload -N` *(see 'Load Indices' on page 47).*

The order string is used for host sorting and selection. The ordering begins with the rightmost index in the order string and proceeds from right to left. The hosts are sorted into order based on each load index, and if more hosts are available than were requested, the JobScheduler drops the least desirable hosts according to that index. The remaining hosts are then sorted by the next index.

After the hosts are sorted by the leftmost index in the order string, the final phase of sorting orders the hosts according to their status, with hosts that are currently not available for load sharing (that is, not in the `ok` state) listed at the end.

Because the hosts are resorted for each load index, only the host status and the leftmost index in the order string actually affect the order in which hosts are listed. The other indices are only used to drop undesirable hosts from the list.

When sorting is done on each index, the direction in which the hosts are sorted (increasing or decreasing values) is determined by the default order returned by `lsinfo` for that index. This direction is chosen such that after sorting, the hosts are ordered from best to worst on that index.

When an index name is preceded by a minus sign '-', the sorting order is reversed so that hosts are ordered from worst to best on that index.

The default sorting order is `r1m:pg`.

## Resource Usage String

This string defines the expected resource usage of the task. It is used to specify job resource reservations.

The syntax of the resource usage string is:

```
res=value[:res=value]...[:res=value][:duration=value][:decay=value]
```

The *res* parameter can be any load index. The *value* parameter is the initial reserved amount. If *res* or *value* is not given, the default is not to reserve that resource.

The *duration* parameter is the time period within which the specified resources should be reserved. It is specified in minutes by default. If the value is followed by the letter 'h', it is specified in hours. For example, '`duration=30`' and '`duration=2h`' specify a duration of 30 minutes and two hours respectively. If *duration* is not specified, the default is to reserve the total amount for the lifetime of the job.

The *decay* parameter indicates how the reserved amount should decrease over the *duration*. A value of 1, '`decay=1`', indicates that system should linearly decrease the amount reserved over the duration. The default *decay* value is 0, which causes the total amount to be reserved for the entire *duration*. Values other than 0 or 1 are unsupported. If *duration* is not specified, *decay* is ignored.

```
rusage[mem=50:duration=100:decay=1
```

The above example indicates that 50MB memory should be reserved for the job. As the job runs, the amount reserved will decrease at approximately 0.5 megabytes per minute until the 100 minutes is up.

## Job Resource Requirement Specification Examples

LSF resource requirement syntax is straightforward to use, even though it may look complicated. The following examples illustrate how to specify a resource requirement when submitting a job to LSF JobScheduler. These examples show the command line interface to better illustrate the concepts involved (although it is easier to use the GUI for job creation).

### Example 1

This example creates a job that requires more than 50MB of swap space and requires an `aix` type of host to run.

```
% bsub -R “swp > 50 && type == aix” myJob
```

The section name “select” can be omitted because the select string is the first string in the resource requirement. We did not specify “order” or “rusage” in this example, which tells LSF JobScheduler to order hosts by default ordering (“r1m:pg”), and not to reserve resources.

### Example 2

This example specifies a shared resource as a resource requirement.

```
% bsub -R “avail_scratch > 200 && swap > 50” myJob
```

Assume that `avail_scratch` is a shared resource for scratch space in a shared file system and all hosts in the cluster have access to the shared scratch space. The job will only be scheduled if the value of the `avail_scratch` resource is more than 200 MB and will go to a host with at least 50MB of available swap space.

### Example 3

It is possible for a system to be configured so that only some hosts within the LSF cluster have access to the scratch space. In order to exclude hosts which cannot access a shared resource, the “defined(resource_name)” function must be specified in the resource requirement string.

```
% bsub -R "defined(avail_scratch) && avail_scratch > 100 && swap > 100" myJob
```

This command will exclude any hosts which cannot access the scratch resource. The LSF administrator configures which hosts do and do not have access to a particular shared resource.

# Configuring Resource Requirements

Some applications require resources other than the default set. LSF can store resource requirements for specific applications so that users do not have to specify resource requirements each time a job is submitted.

## Remote Task List File

The resource requirements of applications are stored in the *remote task list file*. A *task* is a command or a user-created executable program; the terms *application* or *job* are also used to refer to tasks. The *remote task list file* contains the resource requirements of each task.

There are three sets of task list files: the system-wide default file `lsf.task`, the cluster default file `lsf.task.`*cluster*, and the user file `$HOME/.lsftask`. The system and cluster default files are set by LSF administrator and they apply to all users. The user file specifies the tasks to be added to or removed from the system lists for your jobs. Resource requirements specified in your user file override those in the system lists.

When you submit a job to LSF JobScheduler, the system automatically picks up the job's default resource requirement string from the remote task list files, unless you explicitly override the default by specifying the resource requirement string on the command line or job submission GUI.

## Managing Your Task List

The `lsrtasks` command inspects and modifies the remote task list. Invoking `lsrtasks` commands with no arguments displays the resource requirements of tasks in the remote list, separated from the task name by '/'.

```
% lsrtasks
cc/cpu                    compress/-:cpu:mem       compressdir/cpu:mem
f77/cpu                   verilog/cpu && cadence   synopsys/swp >150 && cpu
dsim/type == any          hspice/cpu && cadence    nas/swp > 200 && cpu
cfd3d/type == SG1 && cpu  epi/hpux11 sparc         regression/cpu
cc/type == local          compress/cpu
```

You can specify resource requirements when tasks are added to the user's remote task list. If the task to be added is already in the list, its resource requirements are replaced.

```
% lsrtasks + myjob/swap>=100 && cpu
```

This command adds `myjob`, along with its resource requirement, to the remote tasks list.

# 5. Defining Jobs

A job is a program or a command that is run on a host within an LSF cluster. A job can be a one-time job that is run only once and leaves the system forever, a repetitive job that is run every time the associated dependency conditions are met, or an ad-hoc job that will not run until a user explicitly directs it to.

## Types of Jobs

There are three types of jobs in LSF JobScheduler.

- repetitive
- ad-hoc
- one-time

A job is treated as a *repetitive job* if the job has a dependency condition specified. A job is considered to have a dependency condition if it is associated with an event as described in *Section 3, 'Events and Calendars', beginning on page 21*.

A job is treated as an *ad-hoc job* if it is not a repetitive job and is submitted with a hold requirement. A job with a hold requirement will be suspended as soon as it is created. It must be explicitly resumed before it is considered for scheduling. After an ad-hoc job finishes, it returns to suspended status for reexecution.

*One-time jobs* are jobs that are submitted by users for execution as soon as conditions are right, and then removed from LSF JobScheduler memory without further user involvement. A one-time job is executed once only, does not have a dependency condition, and is not submitted with a hold requirement.

Jobs can be grouped into job groups for easy management. A job group is a container for jobs, similar to the way in which a directory is a container for files. Multiple levels of job groups can be defined to form a hierarchical tree. A job group can contain jobs and sub-groups.

# Job Attributes

A job can have several key attributes:

- **`jobId` -** a positive integer that uniquely identifies the job. Every job in the LSF JobScheduler is automatically assigned a job ID, which is returned by LSF JobScheduler when the job is submitted.

- **`jobName` -** assigned as an additional identifier to simplify reference and manipulation. This name does not have to be unique. If you do not supply a name, the system uses the name of the submitted command as the `jobName`.

- **job group path -** the name and location of a job group within the job group hierarchy. Job groups allow the organization of a collection of jobs into a hierarchical tree similar to the directory structure of a file system.

  In LSF JobScheduler, every job belongs to a job group. If a user does not specify a job group name when submitting a job, the job will be created under the “root” group, denoted as “/”.

- **owner -** the login name of the user who creates the job or job group. Every job or job group must have an owner, who has permission to manipulate the job or job group by performing, for example, deletions, modifications, or sending control signals. The LSF administrator has permission to manipulate the jobs of all users.

# Job Status

After the job is submitted, it is placed into a job queue where it waits to be scheduled by LSF JobScheduler. The job will be automatically started by LSF JobScheduler on a

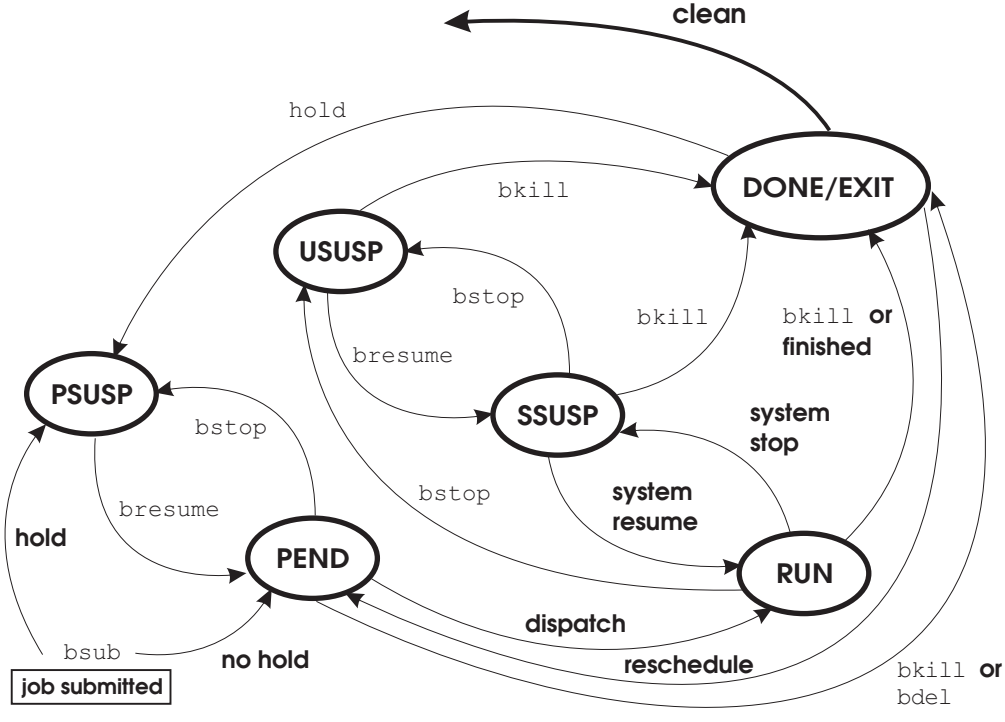
suitable machine in the cluster once the specified conditions are met. After the job has finished, the output from the job is delivered to the user, either into a specified file or via email. If it is a repetitive job, it is placed back into the queue where it waits to be scheduled the next time the specified conditions are met.

*Figure 10* shows the state transitions a job may experience during its life-cycle. LSF JobScheduler maintains and updates the status of each job as it passes into different states. The possible job states are:

- **PEND -** waiting in the queue for scheduling
- **PSUSP -** on hold. The job was submitted with the hold flag or job was suspended by the user while in PEND state.
- **RUN -** dispatched to a host and running
- **SSUSP -** suspended by LSF JobScheduler while running. This happens when the load on the execution host exceeds the configured threshold (as can be seen from the output of the `bhosts -l` command).
- **USUSP -** suspended by the owner of the job or by the LSF administrator while the job is running
- **DONE -** finished execution with a zero exit code
- **EXIT** - finished execution with a non-zero exit code

When a job is submitted, it is given PEND status by default. If the hold flag is specified for the job when it is submitted, the job will be given PSUSP status. Submitting the job into the PSUSP state prevents it from being scheduled until explicitly requested.

Figure 10. Job States




Depending on the nature of the job, a job with DONE or EXIT status is handled differently. If a job is associated with an event that is not a time event, the job will go back into the PEND state immediately, waiting for the event to become active again. If a job is time event dependent, the job will stay in the DONE or EXIT state for a configured period of CLEAN_PERIOD (as can be seen by reading the output of the `bparams` command), or as soon as the time event becomes active again, whichever happens first. Then the job will be requeued with PEND status.

If a job is a one-time job, the job will stay in the DONE or EXIT state for a configured period of CLEAN_PERIOD (as can be seen by running the `bparams` command), and then will be removed from LSF JobScheduler memory.

If a job is submitted with hold flag, the job will go back to the PSUSP state immediately after it finishes.

As can be seen from the state diagram, a repetitive job never leaves the system, unless it is explicitly deleted.

A non-repetitive job submitted with the hold flag will be given PSUSP status when the job is finished. A job with hold status will go into the PEND state only if the user resumes it.

Jobs may also be suspended at any time. A job can be suspended by its owner, by the LSF administrator, or by LSF JobScheduler. There are three different states for suspended jobs: PSUSP, USUSP, and SSUSP.

# Creating a Simple Job

You can use either the LSF JobScheduler GUI or the `bsub` command to submit a job to the system. *Figure 11* shows the Job Submission window of the LSF JobScheduler `xbsub` GUI. All that is required in this window is the actual command line you want to execute. LSF JobScheduler will find a suitable host to run your job if you do not specify one.

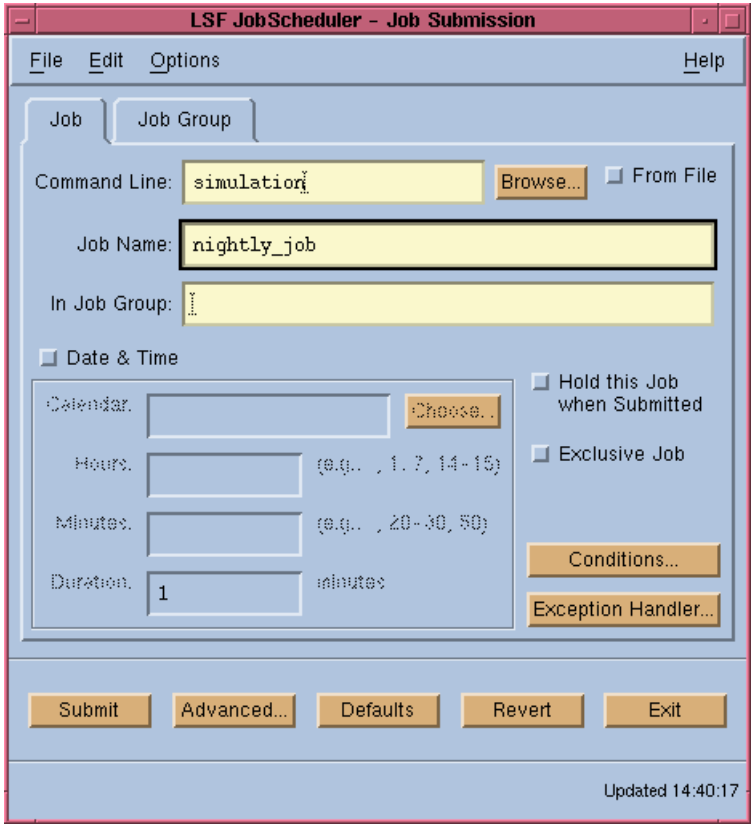
The same job can be submitted to LSF JobScheduler using the `bsub` command:

```
% bsub -J nightly_job simulation
Job <101> is submitted to the default queue <normal>.
```

The `job_name` is a string of text declared with the `-J` option. If the string contains blanks or special characters, it should be placed within quotes. When you submit the

job to the system, a job ID is assigned and displayed. If you do not supply a name, the system uses a portion of the command name as the default job name.

Figure 11. Job Submission Window




Since this job is not associated with a dependency condition, it is a simple one-time job. To define a repetitive job, associate it with a dependency condition. See *'Specifying Dependency Conditions' on page 80* for more information.

# Input and Output

When one of your jobs completes or exits, the system emails you, by default, a job report together with the job's standard output (`stdout`) and error output (`stderr`). The output from `stdout` and `stderr` are merged together in the order in which they were printed, as if the job had been run interactively.

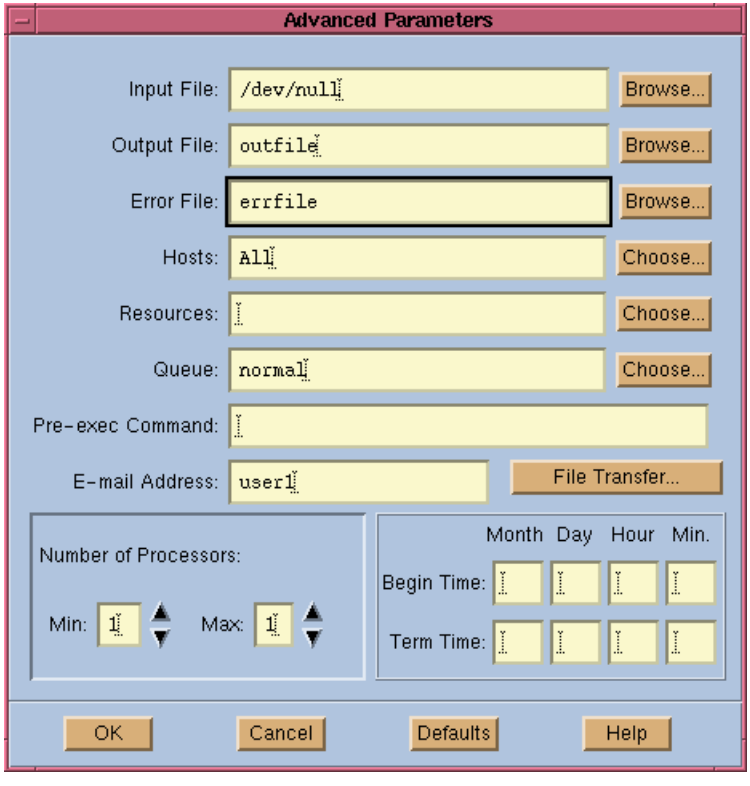
**UNIX** By default, the `stdin` of the job is set to `/dev/null`.

**NT** By default, the `stdin` of the job is set to `NUL`.

If you do not wish to receive email for the `stdout` and `stderr` of your jobs, you can customize this at job submission time. *Figure 12* shows the GUI interface for specifying job parameters—standard input and output and error output are the first three fields in the window. This window is opened by clicking on the "`Advanced`" button of the `xbsub` main window as shown in *Figure 11*.

If you choose to receive email, you can redirect it to a specified user instead of your current login name.

Figure 12. Job Parameters Window




The same result can be achieved via the `bsub` command interface:

```
% bsub -o outfile -e errfile -u user1 -q normal simulation
Job <102> is submitted to default queue <normal>.
```

If you specify the `-o` *outfile* argument but do not specify the `-e` *errfile* argument, the standard output and error are merged and stored in *outfile*.

The output file reported by LSF JobScheduler normally contains job report information as well as job output. This information includes the submitting user and host, the execution host, the CPU time used by the job, and the exit status.

The output files are created on the execution host.

## Host Selection

LSF JobScheduler provides you with many ways to restrict the set of candidate hosts on which your jobs may be run.

Clicking on the "Choose" button beside the "Hosts" area (shown in *Figure 12*), displays a list of all LSF JobScheduler server hosts on which your job may be run. *Figure 13* is an example of a host selection window. Click on "OK" to finish host seletion. All hosts chosen will be displayed in the "Hosts" field of the original window (see *Figure 12*).

The hosts you choose at job submission time are candidate hosts for the job. LSF JobScheduler will use intelligence in determining which host should be used to run the job, depending on the dynamic load situation. If you want to restrict your job to run on one specific host, choose only that host as the candidate host.

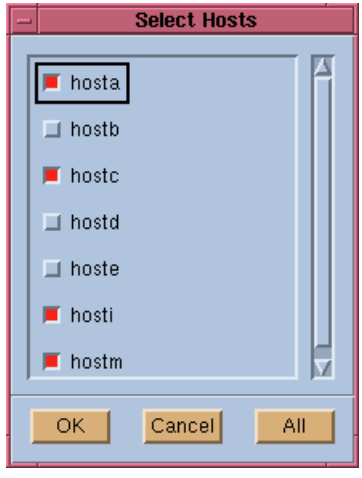
Host selection for a job can also be done using the `bsub` command line:

```
% bsub -m "hosta hostb hostc" simulation
Job <103> is submitted to default queue <normal>.
```

Any host(s) you choose must also satisfy all other scheduling conditions in order to be eligible to run the job.

By specifying more than one host for your job, high availability is achieved automatically, because as long as one of the hosts you specify is up and running, the job will be able to run.

Figure 13. Host Selection Window




## Host Groups

If you have a large cluster with many hosts, it can be inconvenient to type in or select the same or a similar set of hosts if you frequently run jobs that are restricted to the same hosts. To make this easier, LSF JobScheduler allows you to put hosts together in host groups, and then select the name of the group you want, rather than each host individually.

A host group is just an alias for a group of hosts in LSF JobScheduler. To see what host groups are configured by your LSF administrator, run the `bmgroup` command. Host group names can be used in any place that a host name can be supplied as a parameter. For example:

```
% bsub -m HPservers myjob
Job <104> is submitted to default queue <normal>.
```

This submits a job that will run on one of the hosts defined by the host group `HPservers`.

### Host Preference

In some situations, you may want to specify a preference for the hosts chosen, rather than an outright restriction.

For example, you may prefer to run a job on a big server because it is faster. But since that server host may not always be available, you want to specify two other slower hosts as backups in case the big server is not available.

Host preferences can be specified together with hosts chosen. For example:

```
% bsub -m hosta hostb+1 hostc+2 command
```

This tells LSF JobScheduler that the job should be run on *hostc* if it satisfies the requirements. Otherwise, run it on *hostb*. *hosta* should be used only if neither *hostc* nor *hostb* can run the job. The "`+number`" following the host name indicates the preference level of the chosen host.

You can also specify host preferences using the GUI interface.

## Queue Selection

When more than one queue is available, you need to decide which queue to use. If you submit a job without specifying a queue name, LSF JobScheduler chooses a suitable queue as the default queue.

## Specifying the Default Queue

Use the `bparams` command to display the default queue:

```
% bparams
Default Queues: normal
Job Dispatch Interval:20 seconds
Job Checking Interval:80 seconds
Job Accepting Interval:20 seconds
```

This command displays LSF JobScheduler parameters configured by your cluster administrator.

You can override the system default by defining the environment variable `LSB_DEFAULTQUEUE`. For example:

```
% setenv LSB_DEFAULTQUEUE priority
```

## Choosing a Queue

The default queue is normally suitable to run most jobs. If you want to submit jobs to queues other than the default queue, you should choose the most suitable queue for each job.

To specify a queue for your job, simply put a queue name in the "`Queue`" area of the Job Parameter window (shown in *Figure 12*). If you do not know what queues are available, click on the "Choose" button beside the "Queue" field. This displays a popup window from which you can select a queue for your job. It is possible to choose multiple queues for your job, in which case LSF JobScheduler will automatically find a queue that will be able to handle your job, based on your job's parameters.

To see detailed queue information, use the `bqueues` command or use the LSF JobScheduler `xlsjs` GUI.

# Resource Requirements

Resource requirements specify the resources required before a job can be scheduled to run on a host. This is especially useful when your cluster consists of machines with different architectures, operating systems, or hardware/software resources. Resource requirement support is a powerful mechanism for resource mapping in LSF JobScheduler. For background information on resource requirements, See *Section 4, 'Resources', beginning on page 45*.

By specifying resource requirements, your job is guaranteed to run on a host with the desired resources. For example, if your job must be run on a host with the Solaris operating system, you can specify this requirement. LSF JobScheduler will consider only Solaris machines as candidate hosts for your job.

With resource requirements specified for your job, you do not have to specify candidate hosts. You can view your cluster as one virtual machine with different resources. You specify the resource requirements for your job; LSF JobScheduler matches your job's resource requirements to actual resources that are available. For example, if you know your job needs an HPPA machine and at least 50MB swap space to run, simply include "`type==HPPA&&swp>50`" as the job's resource requirement.

You do not have to specify a resource requirement each time you submit a job. Simply put the job's resource requirement in your remote task list so that LSF JobScheduler automatically finds this resource requirement by command name. See *'Configuring Resource Requirements' on page 57* for remote task list operations.

By specifying resource requirements explicitly when you submit a job, you override those defined in your remote task list. If your job's resource requirements are not defined in your remote task list, and you do not specify a resource requirement explicitly at job submission time, LSF JobScheduler assumes the default resource requirement. The default resource requirement is that your job be run on a host of the same type as the host from which the job is submitted.

# Pre-execution Commands

Some jobs require resources that LSF JobScheduler does not directly support. For example, a job may need to successfully create a scratch space before it can run. This pre-execution procedure may or may not succeed, depending on the dynamic situation on the execution host.

A pre-execution command is a job attribute you can specify so that your job will run on a host only if the pre-execution command has successfully completed. The pre-execution command returns information to LSF JobScheduler using its exit status. If the pre-execution command exits with non-zero status, the main job is not dispatched. The job goes back to the `PEND` state and is rescheduled later.

A pre-execution command can be defined in the job parameter window shown in *Figure 12*, or you can use the `-E` option of the `bsub` command. The following example shows a job that requires a tape drive. The program `tapecheck` is a site-specific program that exits with a status of `0` if the specified tape drive is ready, and exits with a status of `1` otherwise.

```
% bsub -E "tapecheck /dev/rmt0l" backup
```

A pre-execution command is executed on the same host as the main job. A pre-execution command is run under the same user ID, environment, and home and working directory as the main job. If the pre-execution command is not in your normal execution path, the full path name of the command must be specified.

The standard input, output and error files for the pre-execution command are those of the main job.

The LSF JobScheduler system assumes the pre-execution command can be run many times without having side effects.

**Note**

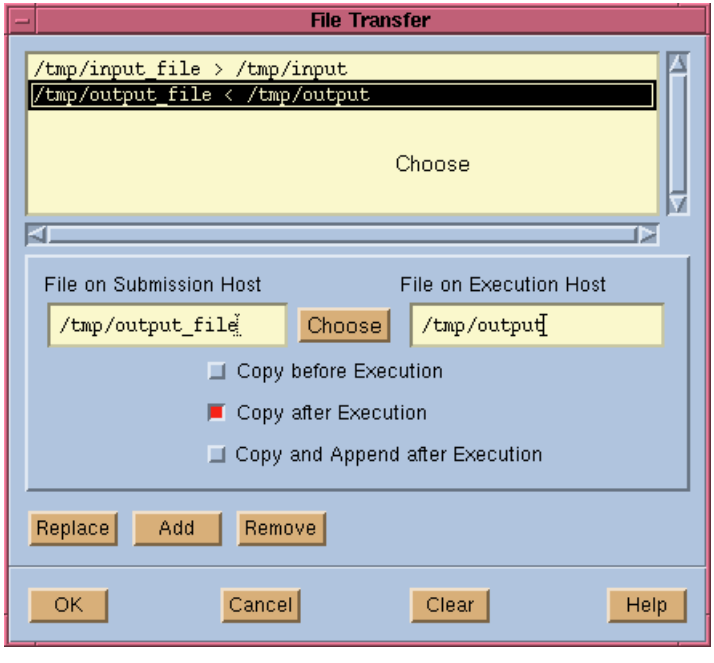
*An alternative to using the `-E` option is for the LSF JobScheduler administrator to set up a queue level pre-execution command. See "Queue-Level Pre-/Post-Execution Commands" in the LSF JobScheduler Administrator's Guide for more information.*

# File Transfer

LSF JobScheduler is normally used in networks with shared file space. When shared file space is not available, LSF JobScheduler can copy needed files to the execution host before running the job, then copy the resultant files back to the submission host after the job completes.

When you click on the “File Transfer” button in the Job Parameter window (shown in *Figure 12*), you will see a window for specifying file transfer requirements (shown in *Figure 14*). You can specify multiple files to be transferred, and in different ways.

Figure 14. File Transfer Requirement Window




File transfer requirements can also be specified using the `bsub` command with the following option:

```
[lfile op [rfile]]
```

**lfile**

This is the file name on the submission host.

**rfile**

This is the file name on the execution host.

The `lfile` and `rfile` parameters can be specified with absolute or relative path names. If you do not specify one of the files, `bsub` uses the file name of the other. At least one must be given.

**op**

This is the operation to perform on the file. `op` must be surrounded by white space, and is invalid without at least one of `lfile` or `rfile`. The possible values for `op` are:

**>**

`lfile` on the submission host is copied to `rfile` on the execution host before job execution. `rfile` is overwritten if it exists.

**<**

`rfile` on the execution host is copied to `lfile` on the submission host after the job completes. `lfile` is overwritten if it exists.

**<<**

`rfile` is appended to `lfile` after the job completes. `lfile` is created if it does not exist.

**><, <>**

`lfile` is copied to `rfile` before the job executes, then `rfile` is copied back (replacing the previous `lfile`) after the job completes (<> is the same as ><).

If you specified an input for your job (see *'Input and Output' on page 65*), and the input file it is not found on the execution host, the file is copied from the submission host using the LSF JobScheduler remote file access facility. It is removed from the execution host after the job finishes.

If you specified output files for standard output and standard error, these files are created on the execution host. They are not copied back to the submission host by default. You must explicitly copy these files back to the submission host.

LSF JobScheduler tries to change directories to the same path name as the directory where you ran the `bsub` command. If this directory does not exist, the job is run in the temporary directory on the execution host.

If the submission and execution hosts have different directory structures, you must ensure that the directory where `rfile` will be placed exists. You should always specify it with relative path names, preferably as a file name excluding any path. This places `rfile` in the current working directory of the job. The job will work correctly even if the directory where the `bsub` command is run does not exist on the execution host.

In addition, you can also specify any files that need to be transferred for the job between the submission machine and execution machine.

# Grouping Jobs

When developing a complex schedule containing many jobs, it is useful to organize related jobs into groups so that it becomes easier to view and manipulate them. LSF JobScheduler allows you to group your jobs into logically related job groups to make your life easier. For example, if your jobs are responsible for the calculation of different portfolios—each portfolio being calculated by a group of related jobs—then you can make every portfolio a job group that contains all jobs responsible for that portfolio. You can then define your schedules at the level of job groups, instead of individual jobs.

LSF JobScheduler supports job grouping where jobs are organized into a hierarchical tree similar to the structure of a file system. Like a file system, the tree contains groups (which are similar to directories) and jobs (which can be considered to be files). Each group can contain other groups or individual jobs. Job groups are created independently of jobs, and can have dependency conditions which control when jobs within the group are considered for scheduling.

The LSF JobScheduler system maintains a single tree under which all jobs in the system are organized. The top-most level of the tree is represented by a group named “/”, the root group. The root group is considered to be owned by the primary LSF Administrator and cannot be removed. Under the root group users can create jobs or new groups. By default, if a user submits a job without a group path, the job belongs to the root group.

## Job Group Status

A job group is a collection of jobs which has a status associated with it. The possible job group status conditions are:

- **active -** a job group has active status if and only if all dependency conditions for the job group are satisfied. Jobs belonging to the job group are considered for scheduling if the job group has active status.

- **inactive -** a job group has inactive status if one or more of the dependency conditions for the job group is not satisfied. Jobs in the job group will not be considered for scheduling if they have inactive status.

- **hold -** on hold. No jobs in the job group will be considered for scheduling regardless of the dependency conditions of the job group.

When a new job group is created, it is automatically given hold status. This allows you to finish building your job group hierarchy and then to release it after it is ready to be scheduled. You must explicitly release a job group in order for jobs in the job group to be scheduled. You can also explicitly give a job group hold status. This allows you to "freeze" the scheduling of the job group. See *'Job Controls' on page 109* for more details.

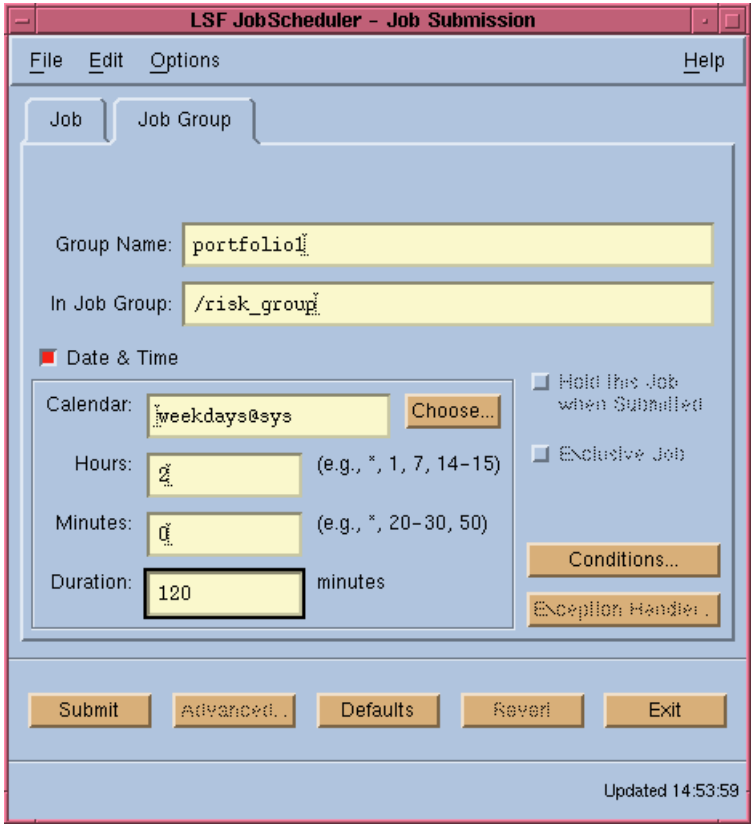
## Creating a Job Group

You can create a new job group from the command line, or from the graphical job submission tool `xbsub`.

To do this using `xbsub`, select the "Job Group" tab, (as shown in *Figure 15*) to display the appropriate options. The job group definition options in this window are a subset of those displayed when the "Job" tab is selected (see *Figure 11*). You can specify a time event dependency and other event dependencies for a job group, just as you can for an individual job.

The “In Job Group” field specifies the parent job group path name starting from “/”. If this parameter is not specified, the new group will be created under the root group.

Figure 15. Job Group Definition Window




Job groups can also be created from the command line. The following examples show how to create a job group:

```
% bgadd /risk_group
```

This creates a job group named “risk_group” under the root group “/”.

```
% bgadd /risk_group/portfolio1
```

This creates a job group named “portfolio1” under job group “/risk_group”.

When creating a group from the command line, you must provide a group specification with a full group path name. The last component of the path is the name of the new group to be created. A parent job group must have been created before you create a sub-group under it.

When the group is initially created it is given HOLD status. The above example creates job groups without dependency conditions. Job groups will always have ACTIVE status once you release them from HOLD status.

Each group is owned by the person who created it. The job group owner can operate on all jobs within the job group and its sub-groups. It is possible for a user to add a job or group into a group that is owned by another user.

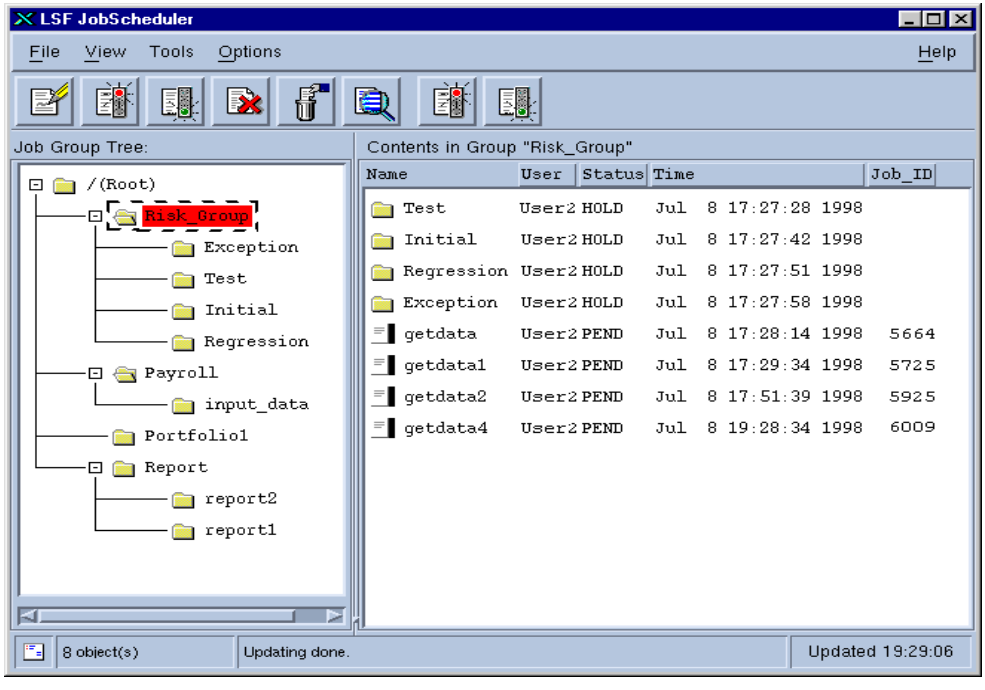
Use the `xlsjs` graphical tool to view job groups and jobs under the job groups. The `xlsjs` tool provides an intuitive interface for monitoring jobs and job groups. *Figure 16* shows the `xlsjs` graphical tool.

The left side of the `xlsjs` GUI displays the job group tree structure. You can expand and shrink the view by clicking on the group names at different levels. The right side of the window displays jobs (upper area) and job groups (lower area) under the current job group. The current job group is highlighted in the job group tree on the left side of the window. The views on the right side can be adjusted to show information that is of interest to you.

The `xlsjs` GUI is the console window of LSF JobScheduler. You can perform almost all LSF JobScheduler-related operations from this window. For example, to create a new job group under “/risk_group”, simply select “/risk_group” and then choose **File | New | Group**. This displays the window as shown in *Figure 15*.

Command-line tools are also available to view and manipulate job groups.

Figure 16. Job and Job Group Monitoring Window




## Submitting a Job under a Job Group

After you have created the desired job groups, you can then submit jobs into them. To submit a job into a job group, select the job group and then choose **File | New | Job**. This displays the job submission window as shown in *Figure 11*, with the current working group set to the selected group.

You can also submit a job into a job group using the `bsub` command. For example:

```
% bsub -J /risk_group/portfolio1/newjob myjob
Job <105> is submitted to default queue.
```

The `-J` option of the `bsub` command, followed by a group path, tells LSF JobScheduler the exact location of the job group the job should belong to. If you assign a unique job name to each job created, it will be easier for you to keep track of your jobs.

When using the command-line job submission tool, `bsub`, the job name parameter should be used to specify the full path of the group in which the job is to be placed.

You will see the submitted jobs on the right side of the window as shown in *Figure 16*. You can also view submitted jobs using the `bjobs` command. With the `-g` option of the `bjobs` command, you can view job groups as well as jobs.

If you do not see submitted jobs in the `xlsjs` window, make sure that you have chosen the right filters by choosing **View | Filter Jobs**. If you are submitting jobs to a newly created job group, remember to release the job group after you submit all jobs so that the job group can enter the ACTIVE state.

# Specifying Dependency Conditions

Because many jobs are operations in response to various events, the scheduling of such jobs is dependent on specific events occurring. A dependency condition is a job or job group attribute you can specify so that your job or job group gets ready to run when certain events happen. A dependency condition can be specified in terms of time events, job events, job group events, exception events, file events and user events. For a conceptual explanation of these events, see *Section 3, 'Events and Calendars', beginning on page 21*.

## Time Event Dependency

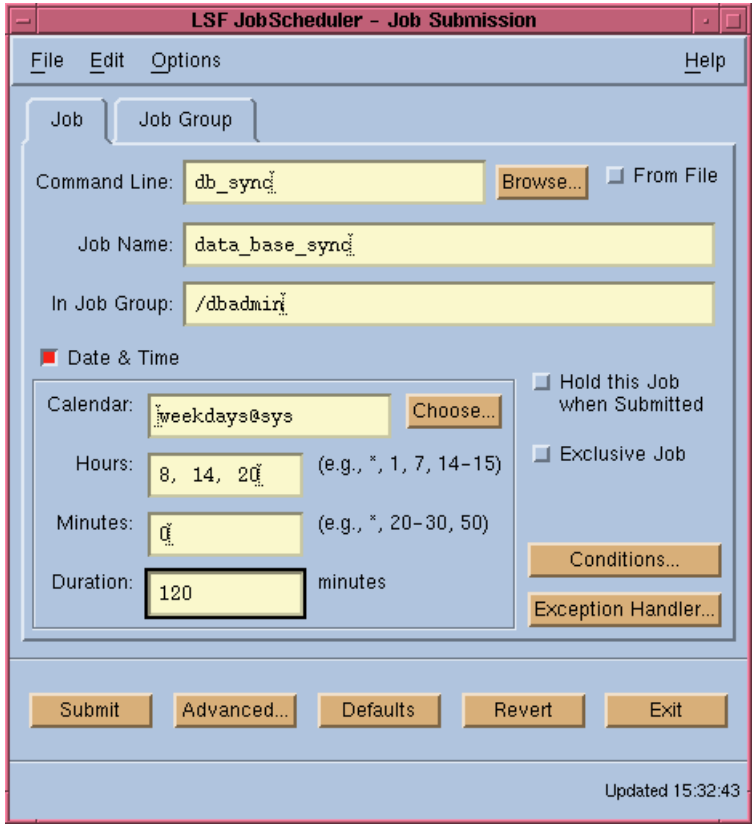
If your job needs to run periodically, or at pre-determined times, you can associate a time event with your job. A time event can be specified at either the `xbsub` GUI or the `bsub` command line level.

To specify a time event for your job, simply enable the "Date and Time" checkbox in the `xbsub` main window. You can then specify the time event, as shown in *Figure 17*. A time event specification contains two parts: a calendar and a time specification. By

clicking on the “Choose” button, you can choose a calendar from all calendars defined in the system. To view calendar details, use the `xbcal` GUI.

Normally, you will choose from your own calendars and the system calendars, but you can also use other users’ calendars. If you do use a calendar defined by another user, remember that the other user’s calendar you are depending on can be modified by its owner without warning!

Figure 17. Specifying Time Events



The calendar specifies the days during which the event will repeat. If you do not specify a calendar in the “Date & Time” area, LSF JobScheduler assumes the default daily calendar, i.e. every day.

### Time Specification

The time specification area specifies the way in which the time event should repeat in a day defined by the calendar. It contains 3 fields: “Hours”, “Minutes”, and “Duration”.

The “Hours” field specifies at which hours during the specified day(s) the event should repeat. You can specify several time points separated by commas, or a range of hours such as 5-17, or a combination of the above. The event will repeat at each hour specified in this field. Valid values for “Hours” are 0-23.

The “Minutes” field specifies at which minutes during the hour(s) specified in the “Hours” field the event should repeat. You can specify several time points separated by commas, or a range of minutes such as 10-30. The event will repeat at each minute specified in this field. Valid values for “Minutes” are 0-59.

The “Duration” field specifies how long the time event remains active after it becomes active, and should be specified in minutes. It is important that you specify a reasonable duration for your time event to allow your job time to be scheduled. The value for this parameter should not exceed its recurrence interval. For example, if the time event happens every eight hours, then the duration should not be more than eight hours. If you specify a duration that is longer than the interval, it is considered to be the same as the interval.

The job will be scheduled only if the time event is active. If the job is not able to run before the time event becomes inactive, the job is considered to have missed its schedule, and an exception will be triggered (if you have configured one). The job will not be run until the next time the event becomes active.

Time events can also be associated with jobs using the `bsub` command line interface. For example:

```
% bsub -T “weekdays:8,14,20:0%120” dbsync
Job <107> is submitted to default queue <normal>.
```

See *'Time Expressions and the Command Line Interface' on page 43* for details of time expression syntax for the command-line interface. To view the calendars in the system, use the `bcal` command. See *'Manipulating Calendars Using the Command Line Interface' on page 40* for an example of `bcal` command output.

## Inter-job Dependencies

Some of your jobs depend on the results of other jobs. For example, a series of jobs could process time sheet data, calculate earnings and taxes, update payroll and tax ledgers, and finally print a cheque run. Most steps can only be performed after the previous step completes.

In LSF JobScheduler, dependencies among jobs are handled by job events. Job events and job status functions are described conceptually in *'Job Events' on page 23* and *'Job Group Events' on page 24*.

A job can also depend on one or more job groups. This is supported by job group events. A job can depend on the status of a job group. A group itself does not execute, but rather the individual jobs under the group. Therefore, the successful completion or failure of a group is determined by the state of the jobs in the group. A set of job group status functions are provided which expose the various job group counters and the group state. The concepts of job group events and job group status functions are discussed in *'Job Group Events' on page 24*.

By associating job status functions and job group status functions with the current job, you can define inter-job dependencies.

To submit a job that depends on prior jobs or job events, click on the "Conditions" button from the job submission main window. This brings up a dependency condition window as shown in *Figure 18*.

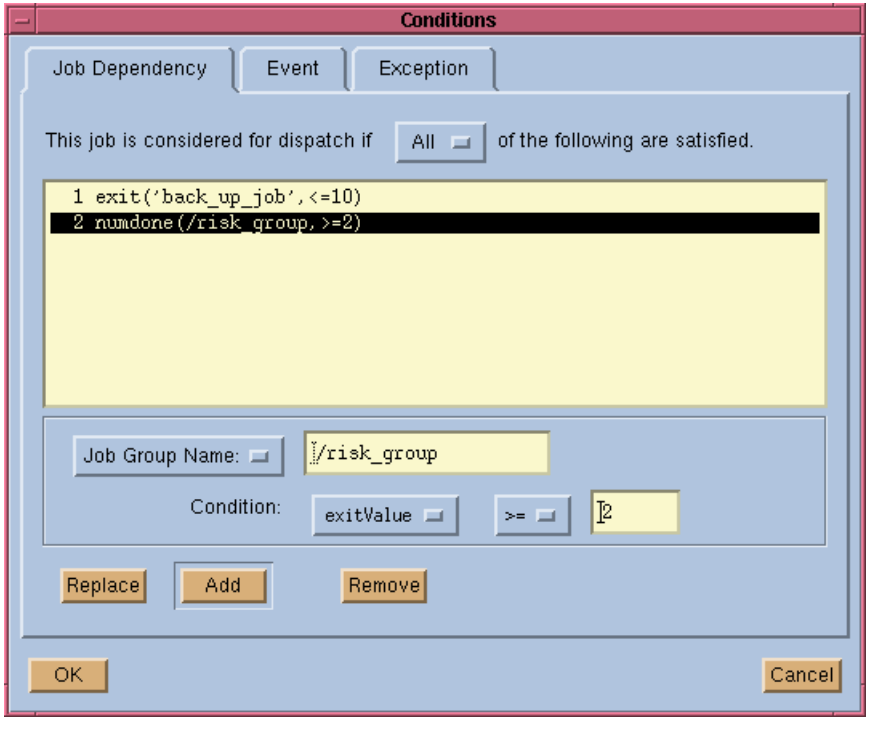
The function `exit(back_up_job)` is a job status function and the function `numdone(/risk_group)` is a job group status function. For a complete list of job status functions and job group status functions, see *'Built-in Events' on page 23*.

As can be seen in *Figure 18*, a job can depend multiple jobs or job groups. In the above example, the dependency condition says the current job will be scheduled when back_up_job has exited with exit code being less than or equal to 10 and when the number of done jobs in job group /risk_group is greater than or equal to 2. Note that you can use either job ID or JobName to specify a job dependency. In any case, the job

or job group to be depended on must already exist before you can create a dependency on it.

A wildcard character '*' can be specified at the end of a job name to indicate all jobs matching the name. For example, `jobA*` will match `jobA`, `jobA1`, `jobA_test`, `jobA.log` etc. There must be at least one match. If more than one job matches, your job will depend on every one of the jobs.

Figure 18. Inter-Job Dependency Condition Window




While `jobId` may be used to specify the jobs of any user, the job name can only be used to specify your own jobs. If you submitted more than one job with the same job name, all jobs with that name are assumed.

The prior jobs are identified by the job ID number or a job name. The job ID is assigned by LSF JobScheduler when the job is submitted. If you did not supply a name during

job submission, the system uses the last 60 characters of the submitted command as the job name.

Inter-job dependency can also be specified at the command level using the `bsub` command. Below are a few examples.

If your job only requires that the prior job has started processing (and it does not matter if it has completed), use the `started` keyword.

```
% bsub -w "started(first_job)" -J second_job time_card
```

If your job requires that the prior job finished successfully, use the keyword `done`.

```
% bsub -w "done(pre_process)" -J main_process cheque_run
```

If your job depends on the prior job's failure (for example, it is responsible for error recovery should the prior job terminate abnormally), use the keyword `exit`.

```
% bsub -w "exit(main_process)" -J error_recovery re_run
```

If your job depends on a particular exit value of another job, the value can be given in the exit function.

```
% bsub -w "exit(main_process,100)" -J error_recovery re_run
```

If the job depends on a range of exit values of another job, the range can be given as:

```
% bsub -w "exit(main_process,< 100)" -J error_recovery re_run
```

When your job only requires that the prior job has finished, regardless of the success or failure (for example, the prior task may end successfully, but with a non-zero exit code), use the keyword `ended`.

```
% bsub -w "ended(cheque_run)" -J clean_up clean
```

**Note**

*If you submit a job that depends on a repetitive prior job, then the newly submitted job also becomes a repetitive job, that is, it will go to the* `PEND` *status after it completes a run instead of being removed from the system.*

Specifying only `jobId` or `jobName` is equivalent to `done(jobId|jobName)`.

A numeric job name should be doubly quoted, for example `-w "'210'"`, since most UNIX shells treat `-w "210"` the same as `-w 210`, causing it to be treated as a `jobId`.

The simplest inter-job dependency condition is a `jobId` or a job name.

```
% bsub -w 8195 jobB
```

Your job may depend on a number of previous jobs. In the example following, the submitted job, `dependent`, will not start until job `312` has completed successfully, and either the job named `Job2` has started or the job named `Job3` has terminated abnormally.

```
% bsub -w "done(312) && (started(Job2) || exit(Job3))" \
-J dependent command
```

The following submitted job will not start until either job `1532` has completed, the job named `jobName2` has completed, or all jobs with names beginning with `jobName3` have finished.

```
% bsub -w "1532 || jobName2 || ended(jobName3*)" -J NumberDepend command
```

If any one of the conditional jobs is not found, the `bsub` command will fail and the job cannot be submitted.

## File Event Dependency

File events monitor the status of files and can be used to trigger the scheduling of your jobs. The concepts of file events and file status functions are discussed in *'File Events' on page 27*. A file event dependency can be specified in logical expressions of file status functions.

With the GUI interface, defining a file event dependency is fairly straightforward. *Figure 19* shows the file event dependency window. This window is brought up when you click on the "Conditions" button from the job submission window as shown in *Figure 11*, and then select the "Event" tab.

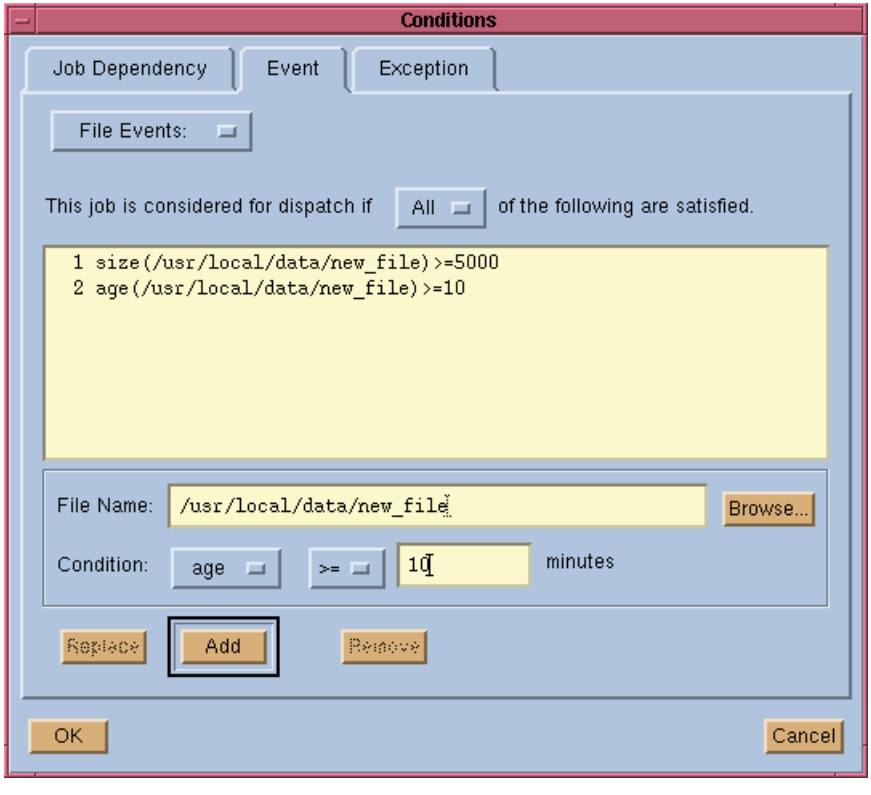
As shown in *Figure 19*, you can specify multiple file event dependency conditions for your job. Note that in the GUI the size parameter of the file is in kilobytes, and the age

parameter is in minutes. Once a job with file event dependency conditions is submitted to the system, LSF JobScheduler will register a file event with the External Event Daemon (`eeventd`) which then monitors the status of the specified file periodically. Once the status of the file event changes, the `eeventd` will inform LSF JobScheduler about the change.

In the example shown in *Figure 19*, the dependency condition is considered satisfied if and only if all of the event conditions listed evaluate to TRUE. You can also specify that the dependency condition be satisfied if any of the event conditions listed evaluates to TRUE.

For a complete list of all available file status functions, see *'File Events' on page 27*.

Figure 19. File Event Dependency Window



A file event dependency condition can also be specified when you submit a job using the bsub command line with the "-w" option and the file keyword. Here are a few examples.

```
% bsub -w "file(age(/u/db/datafile) > 2H)" command
```

This creates a job that runs when the file /u/db/datafile is more than 2 hours old. Note that "H" here stands for hour. Other characters that you can use to represent a time duration include D (day) and W (week). The default is M (minute).

If you want to trigger the job execution by the creation of a file, use the arrival() function. This function detects the transition of the specified file from non-existence to existence.

```
% bsub -w "file(arrival(/usr/data/newfile))"  -R "type==hppa" command
```

This creates a job that runs when file newfile is detected in /usr/data directory. Also note that a resource requirement is specified so that this command should only be run on an hppa host.

Unlike the age() function, the arrival() function does not need a relational operator because the function evaluates to either TRUE or FALSE.

If you are only interested in the existence of the file instead of the transition of the creation, you can use the exist() function.

```
% bsub -w "file(!exist(/usr/data/lock_file))" command
```

This tells LSF JobScheduler to run the job if file /usr/data/lock_file does not exist.

Use the function size() if you want to run a job when the size of the file becomes a certain value.

```
% bsub -w "file(size(/var/adm/logs/log_file) >= 3.5 M)" command
```

The character M refers to megabytes. You could also use K to refer to kilobytes. The default is bytes. Like the age() function, the size() function also requires a relational operator to form a logical expression that evaluates to either TRUE or FALSE.

The file event you are depending on may be on another host.

```
% bsub -w “file(exist(hostd:/usr/local/fileA))” command
```

You can submit a combination of functions. The evaluation of the statement depends on the operators you use. In the following statement, the command will be run if either `fileA` exists or `fileB` arrives (is created).

```
% bsub -w "file(exist(/usr/data/fileA) || arrival(/usr/data/fileB))" \
      command
```

The following statement will evaluate to `TRUE` only if `fileA` exists and `fileB` has arrived.

```
% bsub -w "file(exist(/usr/data/fileA) && arrival(/usr/data/fileB))" \
      command
```

The following command will be run if `fileA` exists and its size is greater than or equal to 1MB.

```
% bsub -w "file(exist(/usr/data/fileA) && \
      size(/usr/data/fileA) >= 1M)" command
```

**Note**

*You must specify the absolute path name of the file in a file status function.*

## Job Exception Event Dependency

A job can be triggered by an exception condition of another job. The concept of job exception events is discussed in *‘Job Exception Events’ on page 25*.

The job exception dependency can be specified either at the command line using bsub or using the job submission GUI. Below is a command line example,

```
% bsub -w “exception(event_name)” recoveryjob
```

This creates a job that will respond to the job exception event `event_name`. By specifying the “`exception`” keyword, you register a job exception event into LSF JobScheduler which monitors the status of this event. `event_name` is an arbitrary string specified by the user.

The event specified here will remain inactive until it is set to active by a real exception from another job. To do so the other job must be submitted with an exception handler to explicitly set the exception event when the exception happens. This can be done by using the `setexcept` action as its exception handler. When an exception handler sets the exception event to active, this triggers all jobs waiting on the job exception event.

Exception handling is discussed in greater detail in *'Exception Handling and Alarms' on page 123*.

## User Event Dependency

You should read this section only if your cluster administrator has installed site-specific event detection functions into the External Event Daemon (`eeventd`).

The concept of user events is described in *'User Events' on page 28*. You can only use the valid user event functions defined for your site.

A user event is created when submitting a job using the `event` keyword. For example, you want to define a user event to detect the status of a tape device before a backup job starts. If the status of the tape device is `READY`, the event becomes `active`, otherwise it is always `inactive`. You can submit the following command:

```
% bsub -w "event(tape_ready)" BackUp
```

A user event, `tape_ready`, is registered by LSF JobScheduler into the External Event Daemon (`eeventd`), which then monitors the event. The string `"tape_ready"` is passed to the `eeventd` by the master scheduler (`mbatchd`). The `eeventd` is responsible for interpreting the string passed to it and must be able to associate the event string passed to it with the actual device or event on which you are dependent.

The above example is a simple one in which the string passed to `eeventd` is a simple string. In fact, your site can have complex syntax defined within the string to provide more sophisticated event status functions, in which case, you must follow the semantics defined by your site in specifying the event dependency condition.

**Note**

*The External Event Daemon (`eeventd`) is a site-specific daemon that is customized and installed by the LSF JobScheduler administrators. See "External Event Management" in the LSF JobScheduler Administrator's Guide.*

## Combining Dependency Conditions

You can submit a job with a combination of conditions. Simply specifying all needed dependencies from the GUI will allow the job to depend simultaneously on time events and multiple other events.

At the command line, use the `-T` and `-w` options of the `bsub` command to specify dependency conditions. The evaluation of the statement depends on the logical expressions you specify. For example:

```
% bsub -w “done(jobA) && file(exist(fileA))” -J jobB command
```

The above statement will evaluate to `TRUE` if `jobA` has completed successfully and `fileA` exists.

## Synchronizing Dependent Jobs

You can synchronize jobs by running the first job from a calendar and submitting the second job to be dependent on the successful completion of the first.

```
% bsub -T “00:00” -J jobA command
Job <8085> is submitted to default queue <default>.

% bsub -w “done(jobA)” -J jobB command
Job <8086> is submitted to default queue <default>.
```

In the above example, jobB will be run every time jobA completes successfully. Since jobA is a repetitive job, jobB also becomes repetitive because of the dependency. If jobA is modified to follow a different calendar, jobB will still run after jobA.

# Other Job Parameters

There are a few other parameters you can specify for your job to further tune the behaviour of your jobs and schedules.

## Number of Processors for Parallel Jobs

If your job is a parallel application, you can also specify the number of processors your job requires to run. You can either choose a range of numbers or a single number. If you choose a range, LSF JobScheduler will schedule the job as long as the number of available processors meets the minimum number. In this case, your parallel application must be able to run with a varying number of processors. If your application has a fixed parallelism, choose a single number, in which case LSF JobScheduler will run your parallel job with exactly that number of processors. This parameter can be specified in the GUI as shown in *Figure 12*, or you can also specify it from the `bsub` command line using the `-n` option.

## Start Time and Termination time

You can also choose start and termination time ranges for your job. Your job will not start until after its start time and will be terminated and removed from the system when the termination time is reached. The start and termination times define your job's life. You do not have to specify both start time and termination times. If a start time is not specified, the default is any time. If a termination time is not specified, the default is never.

Note the difference between the start time / termination time pair and a time event that has a start time and duration. A time event specifies a duration in which the job should be scheduled. The job does not have to finish within the time event duration. A job can only run once for each time duration. The start time and termination time of a job specifies the active life time of a job. The job can run many times within the time range and the job will be terminated and removed from the system when termination time arrives.

The start time and termination time of a job can be specified as shown in *Figure 12*, or from the `bsub` command line using options `-b` and `-t`.

## Exclusive Job

An exclusive job is a job that runs on its own on a machine. LSF JobScheduler will not mix an exclusive job with other jobs. You can define an exclusive job if you want guaranteed performance for that job. Click on "Exclusive" in the job submission window, as shown in *Figure 11*, to submit the job as an exclusive job. You can also do this from the `bsub` command line using the `-x` option.

## Ad-hoc Jobs

A job can be submitted so that it is suspended until it is explicitly resumed by the user or administrator. This type of job is referred to as an ad-hoc job. It is put into the PSUSP state as soon as it is submitted, and a user must resume the job explicitly before it can run. After completion, the job is put back into the PSUSP state waiting for the next run.

Use the `-H` option of the `bsub` command to submit an ad-hoc job from the command line. The `bresume(1)` command causes the job to go into the PEND state, from which it can be scheduled. You can also submit an ad-hoc job from the Job Submission window of the LSF JobScheduler `xbsub` GUI, shown in *Figure 11 on page 64*. Simply enable the "Hold this Job when Submitted" checkbox before clicking on the "Submit" button.

After completion, the job is put back into the PSUSP state waiting for the user to run it once again.

Submitting an ad-hoc job is a useful solution whenever you have a job you need scheduled only when a user requests it. For example, the lead job in a complex schedule of dependent jobs may be an ad-hoc job. Whenever the lead job is released and run, the downstream jobs are triggered. Every time you want to execute the schedule, you need only release the lead job.

## Exception Handlers

You can define exceptions for your job and associate exception handlers to process the exceptions automatically. Exception handlers can be specified at job submission time as shown by the "Exception Handler" button in *Figure 11 on page 64*. This topic will be addressed in greater detail in *'Exception Handling and Alarms' on page 123*.

# 6. Managing Jobs and Schedules

LSF JobScheduler provides a single system image for your cluster so that you can use the whole cluster as if it were a single computer. As such, you can monitor, control, and manipulate your jobs, job groups, and schedules from any host in the cluster through a uniform interface.

The `xlsjs` GUI as shown in *Figure 11 on page 64* is the focal point of all operations. If you do not have any job group defined in the system, you will be seeing only the root job group. All jobs submitted without a job group specification are in the root job group.

If your system has job groups defined by users, you can use `xlsjs` to walk through the job group tree to see the jobs and sub-groups at each level.

You can also manipulate your jobs and job groups through LSF JobScheduler command-level tools.

## Viewing Details of a Job or Job Group

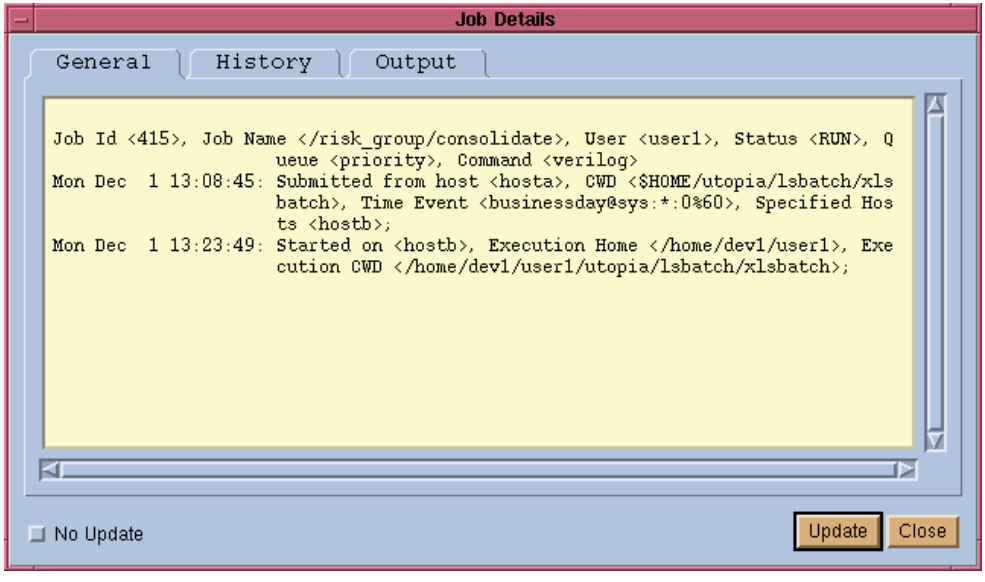
As you walk through the job group tree, you can view all jobs and their status. To view the details of a particular job, double click on the job to get a job detail window, as shown by an example in *Figure 20*.

Depending on the current job status, the job detail information can be different. If the job is already running, the details will contain information about the job execution, such as execution host, process IDs of the job, resources used so far, and current working directory on the execution machine.

If the job is currently in PEND status, the job detail information will tell you why.

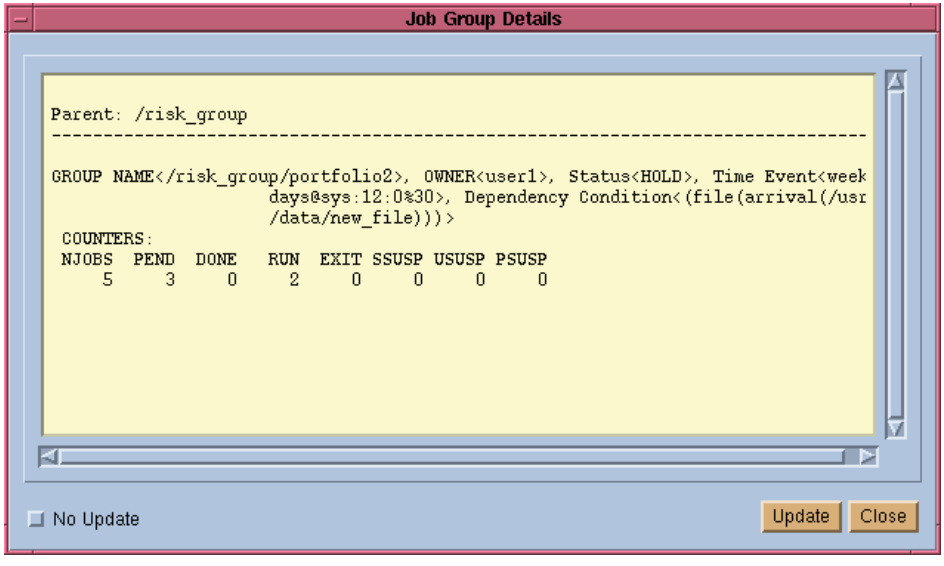
For suspended jobs, the job detail information will include why the job was suspended.

Figure 20. Job Details Window




To view the details of a job group, select the job group from the `xlsjs` GUI and then choose **File | Details**. You can also achieve this by double-clicking on the group listed on the right hand side of the window. The job group details will be displayed in a popup window as shown in *Figure 21*.

Figure 21. Job Group Detail Window




Job and job group information can also be viewed from the command line using the `bjobs` command. For example:

```
% bjobs -a
JOBID USER  STAT QUEUE     FROM_HOST EXEC_HOST JOB_NAME   SUBMIT_TIME
1031  user1 PEND priority  hostc               /sleep 567 Nov 27 16:23
887   user1 DONE priority  hostc     hostb     *nsolidate Nov 27 15:36
1006  user1 DONE priority  hosta     hostb     *nsolidate Nov 27 16:00
```

Note if you do not specify the `-a` option, jobs having DONE and EXIT status will not be displayed. By default `bjobs` will only display the jobs you submitted. Use the `-u user_name` option to view the jobs of other users. Use the reserved user name `all` to see the jobs of all the users.

```
% bjobs -u all
JOBID  USER  STAT  QUEUE    FROM_HOST EXEC_HOST  JOB_NAME   SUBMIT_TIME
6745   user2 RUN   business hostd     hostb      report     Dec 19 09:04
6916   user3 RUN   business hosta     hostd      analyse    Dec 19 09:05
6848   user1 PEND  sysadm   hosta                diskcheck  Dec 17 11:52
```

```
7142   user1 PEND  sysadm   hosta                 backup     Dec 21 15:45
7157   user4 PEND  night    hosta                 forecast   Dec 18 10:56
```

Use the `-s` option to view the suspended jobs only. Along with the job information, it also shows the reason why the jobs are suspended.

```
% bjobs -s
JOBID USER   STAT   QUEUE   FROM_HOST  EXEC_HOST  JOB_NAME  SUBMIT_TIME
1999  user1  PSUSP  default hosta                joba      Dec 10 15:33
The job was suspended by user or system admin while pending;
```

Use `-p` option to view the pending jobs only. Along with the job information it also shows the reason why each job was not dispatched during the last dispatch turn.

```
% bjobs -p
JOBID USER  STAT   QUEUE    FROM_HOST  EXEC_HOST  JOB_NAME  SUBMIT_TIME
1999  user1 PSUSP  default  hosta                joba      Dec 10 15:33
The job was suspended by user or system admin while pending;
5518  user1 PEND  default   hosta                jobb      Dec 14 10:27
Job dependency condition not satisfied;
8056  user1 PEND  default   hostA                jobb      Dec 20 11:41
Job dependency condition not satisfied;
```

To get the details of your job, use the `-l` option of the `bjobs` command:

```
% bjobs -l -J /risk_group/consolidate
Parent: /risk_group
Job Id <887>, Job Name </risk_group/consolidate>, User <user1>, S
                      tatus <RUN>, Queue <priority>, Command <myj
                      ob>
Thu Nov 27 15:36:39:  Submitted from host <hosta>,CWD <$HOME>, Ex
                      clusive Execution, Requested Resources <hpu
                      x>, Time Event <businessdays@sys:*:0%60>;
Thu Nov 27 15:37:15:  Started on <hostc>;
Thu Nov 27 15:41:53:  Resource usage collected. MEM: 440 Kbytes;
                      SWAP: 1 Mbytes PGID: 21699; PIDs: 21699
```

You can get the same output by running the `bjobs` command with `jobId` as the parameter instead of job group path name.

You can view job group information from the command line, too. This is supported by the `-g` option of the `bjobs` command. For example,

```
% bjobs -g
Parent: /
GROUP      STAT     OWNER  NJOBS  PEND DONE  RUN EXIT SSUSP USUSP PSUSP
fund1_grp  ACTIVE   user1  5      4    0     1   0    0     0     0
fund2_grp  ACTIVE   user1  11     2    3     5   1    0     0     0
bond_grp   HOLD     user4  2      2    0     0   0    0     0     0
risk_gr*   ACTIVE   user2  2      1    1     0   0    0     0     0
admi_grp   INACTIVE user3  4      4    0     0   0    0     0     0

JOBID USER  STAT  QUEUE   FROM_HOST EXEC_HOST JOB_NAME     SUBMIT_TIME
1031  user5 PEND  normal  hostd               /simulation  Nov 27 16:23
```

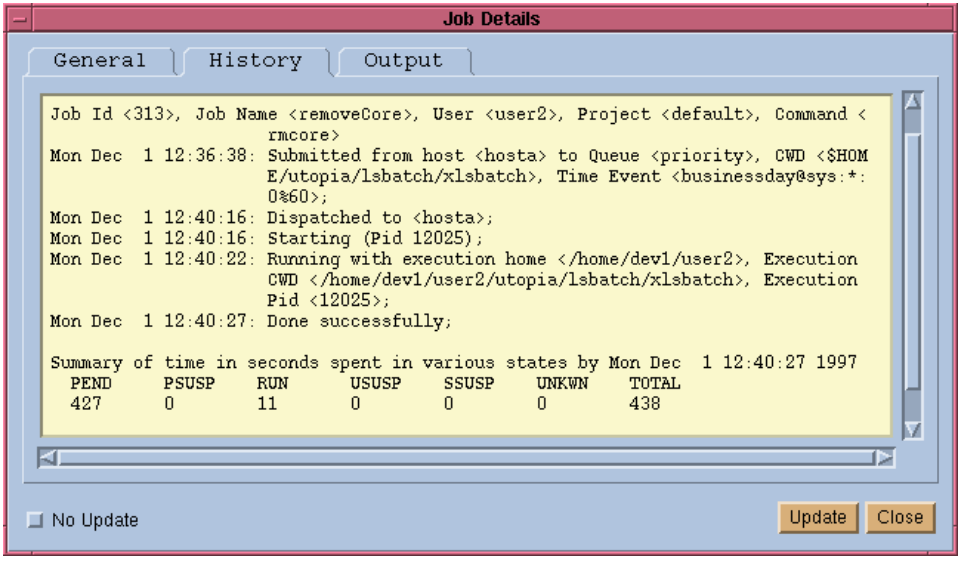
The `-g` option makes the `bjobs` command display job group information as well as job information. It is similar to a directory listing in a file system. By default, the `bjobs -g` command lists all job groups under the Root job group together with all jobs directly submitted into the Root (/) job group. If you specify a `-J /a/b/` group path together with the above command, it will display all job groups under the group `/a/b` together with all jobs submitted to the `/a/b` job group level.

The job counters listed from the above output give a summary of jobs inside the job group tree by different status. If you want to see all job groups and all jobs at all levels, use "`-R`" option together with "`-g`" option of the `bjobs` command.

# Viewing Job History

You may need to know what has happened to your job since it was submitted. By clicking on the "History" tab of the Job Details window, you can see the history of your job, as shown in *Figure 22*. Job history tells you in chronological order what has happened to your job. It also gives you statistics about how long the job has stayed in each job state cumulatively.

Figure 22. Job History Window




The `xlsjs` GUI allows you to see the history of jobs currently in LSF JobScheduler. If you want to see the history of a job that was already deleted from LSF JobScheduler, you must use the `bhist` command.

For example:

```
% bhist -l 7848
Job Id <7848>, Job Name <diskcheck>, User <user1>,Command <find -
                        name core -atime +7 -exec rm {} \;>
Tue Dec 16 11:52:13:  Submitted from host <hostA> to Queue <defau
                      lt>, CWD <$HOME>, Dependency Condition <cal
                      endar(weekdays)>;
Sat Dec 20 07:00:12:  Started on <hostA>, Pid  <29027>;
Sat Dec 20 07:00:12:  Running with execution home </home/user1/>,
                      Execution CWD </home/user1>;
Sat Dec 20 07:00:55:  Done successfully. The CPU time used is 12.
                      .2 seconds;
Sun Dec 21 07:00:05:  Started on <hostA>, Pid <986>;
```

```
Sun Dec 21 07:00:05:  Running with execution home </home/user1>,
                      Execution CWD </home/user1>;
Sun Dec 21 07:01:18:  Done successfully. The CPU time used is
                      11.9 seconds;
Mon Dec 22 07:00:02:  Started on <hostA>, Pid <2892>;
Mon Dec 22 07:00:02:  Running with execution home </home/user1>,
                      Execution CWD </home/user1>;
Mon Dec 22 07:01:13:  Done successfully. The CPU time used is 10.
                      5 seconds;
Tue Dec 23 07:00:10:  Started on <hostA>, Pid <4905>;
Tue Dec 23 07:00:10:  Running with execution home home/user1>, Ex
                      ecution CWD </home/user1>;
Tue Dec 23 07:03:31:  Done successfully. The CPU time used is 19.
                      7 seconds;
Tue Dec 23 15:17:14:  Delete requested by user or administrator <
                      user1>;
Tue Dec 23 15:17:14:  Exited. The CPU time used is 0.0 seconds.

Summary of time in seconds spent in various states by Tue Dec 23 15:17:14 1997
  PEND     PSUSP    RUN    USUSP    SSUSP    UNKWN    TOTAL
  617057   0        44     0        0        0        617101
```

LSF JobScheduler keeps job history information after the job completes a run, so you can look at the history of jobs that ran in the past. The length of the history depends on how often your LSF administrator cleans up old log files. Unless a job ID or a time range (using `-S`, `-C`, `-D`, or `-T`) is specified, `bhist` only displays recent job history, i.e. history stored in the current event log file.

The `bhist` command also allows you to display the history of all jobs chronologically rather than on a per job basis. This gives you a real trace of what exactly happened in the whole system. This is done through the `-t` and `-T` option of the `bhist` command:

```
% bhist -t -T "1997/9/22/18:00,1997/9/22/19:00"
Mon Sep 22 18:50:00:  Job <429> Pending: Job has been requeued;
Mon Sep 22 18:50:00:  Job <429> Dispatched to <hostb>;
Mon Sep 22 18:50:01:  Job <429> Starting (Pid 24289)
Mon Sep 22 18:50:01:  Job <429> Running with execution home </hom
                      e/dev2/user1>, Execution CWD </home/dev2/us
                      r1>, Execution Pid <24289>, Execution user
                      name <user1>;
```

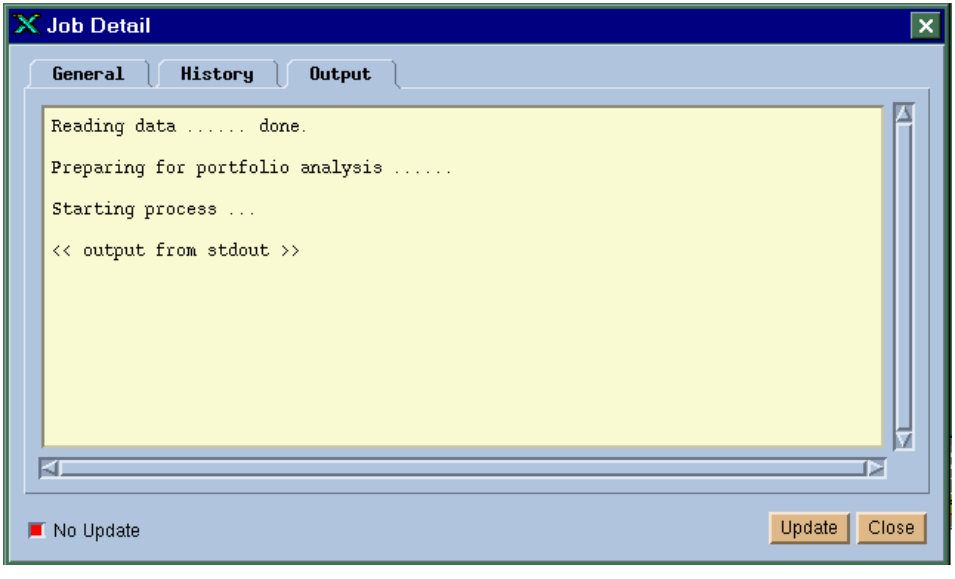
```
Mon Sep 22 18:52:36:  Job <750[19]> Pending: Waiting for scheduli
                      ing after resumed by user;
Mon Sep 22 18:52:52:  Job <750[19]> Dispatched to <hostc>;
Mon Sep 22 18:53:16:  Job <750[19]> Starting (Pid 24708)
Mon Sep 22 18:53:17:  Job <750[19]> Running with execution home <
                      /home/dev2/ussr2>, Execution CWD </home/dev
                      2/user2>, Execution Pid <24708>, Execution
                      user name <user2>;
Mon Sep 22 18:53:23:  Job <429> Done successfully;
Mon Sep 22 18:53:56:  Job <750[19]> Done successfully;
```

This displays job history between 6 PM to 7 PM of September 22. If you specify `-t` but not `-T`, `bhist` assumes the time range of from one week ago to now.

# Peeking at Job Output

You can view the output from the standard output and standard error while the job is running. This can be done through the GUI by clicking on the “Output” tab of the window shown in *Figure 20 on page 96*.

Figure 23. Job Output Window



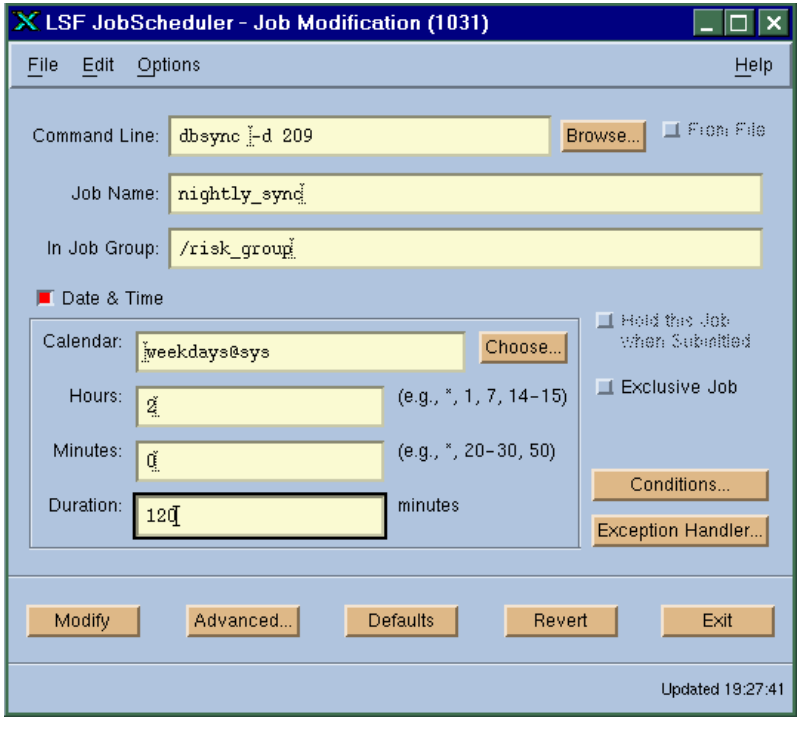

You will only be able to see the output of your own jobs. This is only possible while your job is in RUN, SSUSP, or USUSP status. If the job has not been started or has finished, you cannot see the output. If your job is already finished, the output should be already in the output file as specified in the job submission. See *‘Input and Output’ on page 65* for details.

You can also view the output of your job using the `bpeek` command.

# Modifying a Job

From the `xlsjs` GUI shown in *Figure 16 on page 79*, you can perform different operations on jobs and job groups. You can modify the attributes of a submitted job by selecting the job and then choosing **File | Modify**. You can also do this by selecting the job and then clicking on the job modification toolbar. This will bring up the Job Modification window, as shown in *Figure 24*.

Figure 24. Job Modification Window




The job modification window is almost the same as the job submission window, except that it pre-loads the existing job parameters from LSF JobScheduler. This allows you to make changes to any parameters. After you finish the changes, click on the “Modify” button to commit the changes into LSF JobScheduler.

Job modification will have an effect only on future executions of the job. If the job is running while you do the modification, the current execution will still use the original parameters.

Job modification can be done at the command level using the `bmod` command. The `bmod` command has a set of options similar to those of the `bsub` command. The value for the option you want to modify is overridden with a new value using the same option syntax as the `bsub` command. However if you want to modify the actual command line for the job, you need to use the `-Z` option of the `bmod` command. For example,

```
% bmod -Z "new_cmd arg" 848
Parameters of job <848> are being changed
```

The argument for `-Z` must be enclosed in quotes if it contains more than one string.

It is a little bit more complicated if you use `bmod` to reset an option to its default value. Use the option string followed by 'n'. No value should be specified when resetting an option. For example,

```
% bmod -Tn 848
```

This removes the time event dependency for job 848.

You can only modify jobs owned by yourself and other users; jobs submitted under job groups owned by you. LSF administrator can modify jobs of all users.

# Modifying a Job Group

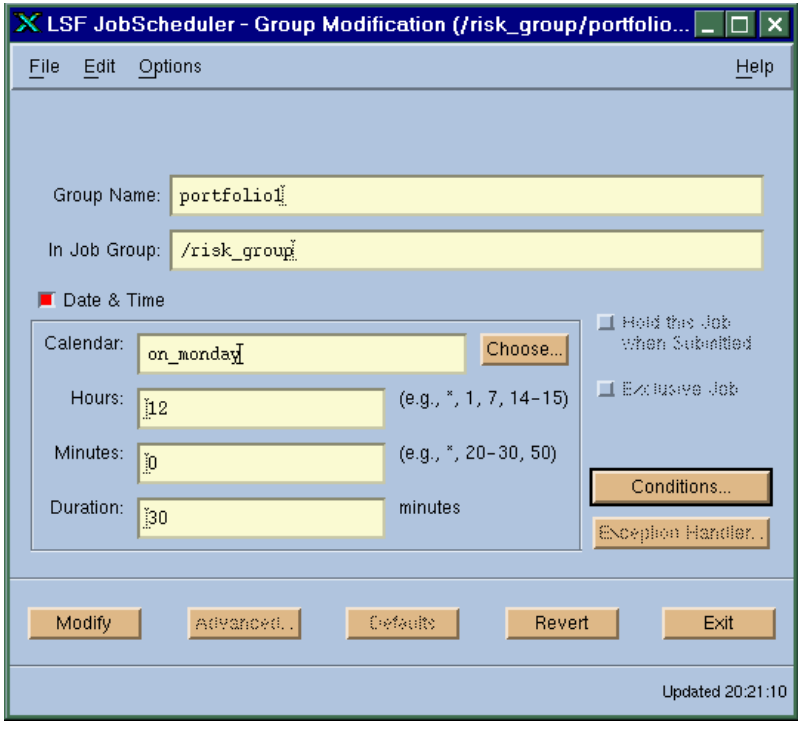
Job groups can be modified in a way similar to that in which jobs are modified. Since the dependency conditions specified at the job group level affect all jobs and job groups in the job group tree, by modifying job group parameters you effectively modify the scheduling behaviour of all jobs belong to the group tree.

*Figure 25* shows the job group modification window. This window appears when you select the group from the `xlsjs` GUI and then choose **File | Modify**.

As with job modification, modification to a job group has no immediate effect on jobs that are already running. It only affects the future scheduling of jobs. However, changing the dependency conditions of a group can cause the status of a job group and all sub-groups to change. For example, if you change the time event dependency of a group and the time event is currently inactive, then the status of the job group and all its sub-groups which have ACTIVE status will immediately become INACTIVE. Jobs already started will continue to run, but jobs that are not scheduled yet will be prevented from further scheduling until the status of the job group becomes ACTIVE again.

You can modify your own job groups and other users' job groups that are created under job groups owned by you. The LSF administrator can modify job groups of all users.

Figure 25. Job Group Modification Window



# Deleting a Job or Job Group

By deleting a job, the job is removed from the memory and the job becomes history. You will be able to see the deleted job from the job history by running `bhist` command.

If the job is currently running, deleting a job kills the job before removing the job from the system. If the job has any events associated with it, these events will also be removed from the system unless these events are still in use by other jobs or job groups.

You can delete jobs or job groups owned by yourself and jobs and job groups that are created by other users underneath a job group owned by you. The LSF administrator can delete jobs or jobs groups owned by all users.

## Deleting Jobs

A job can be deleted easily using the GUI. Click on the job to be deleted and choose **File | Delete**. Or you can click on the job deletion toolbar after selecting the job.

You can also use the `bdel` command to remove a job.

```
% bdel 3456
Job <3456> is being deleted
```

You can specify a job by name using the `-J` option.

```
% bdel -J jobA
Job <3457> is being deleted
```

To delete all jobs directly underneath a job group tree, specify the group path with the `bdel` command:

```
% bdel -J /risk_group/
Job <1202> is being deleted
Job <1203> is being deleted
Job <1204> is being deleted
```

To delete all jobs in a group recursively, use the `-R` option:

```
% bdel -R -J /risk_group/
Job <1202> is being deleted
Job <1203> is being deleted
Job <1204> is being deleted
Job <1205> is being deleted
Job <1206> is being deleted
```

Note that the "/" following the group path is necessary. If the group path is not followed by a "/", the last component in the path is considered as a job instead of a job group.

To remove all jobs inside a job group tree recursively at all levels, use the `-R` option of the `bdel` command.

## Deleting Job Groups

A job group can be deleted from the GUI by selecting the job group and choosing the "Delete" from the "`Group`" pull-down menu. By deleting a job group, you delete all jobs and job groups under the current job group. All events associated with the entire job group tree will be removed as well unless they are also associated with other jobs or job groups.

To remove a job group from the command level, you must first remove all jobs inside the job group tree by running `bdel` command with option `-R`, then remove the job group tree with the `bgdel` command.

```
% bdel -R -J /risk_group/
Job <1202> is being deleted
Job <1203> is being deleted
Job <1204> is being deleted
Job <1205> is being deleted
Job <1006> is being deleted

% bgdel /risk_group
Job group /risk_group is deleted.
```

### Delayed Deletion of a Job

If you want to delete your job after certain a number of runs, use the `-n` option of the `bdel` command. This allows you to specify the number of times your job will execute before it is deleted. After the job runs for the specified number of times, it will be deleted from the system. For example,

```
% bdel -n 5 -J jobA
Job <8087> will be deleted after running next 5 times
```

## Job Controls

After the jobs have been created, you can control their execution and scheduling using LSF JobScheduler's user interface tools. You can control jobs or job groups owned by yourself and jobs and job groups that are created by other users underneath a job group owned by you. The LSF administrator can control jobs or jobs groups owned by all users.

### Terminating a Job

Terminating a repetitive job kills the current run, if the job has been started, and requeues the job. If the repetitive job is in PEND or PSUSP status, i.e., not running, termination has no effect. However, if the job is not a repetitive job, terminating the job has the same effect as deleting the job.

On UNIX, termination sends a sequence of signals to the job in the following order: SIGINT followed by a 10 second delay, then SIGTERM followed by another 10 second delay, and then SIGKILL. On Windows NT, job control messages replace the SIGINT and SIGTERM signals, and termination is implemented by the `TerminateProcess()` system call. The 10 second delays are configurable by your LSF administrator. See the *LSF JobScheduler Administrator's Guide* for details.

You can only terminate your own jobs, or jobs of all users submitted underneath job groups owned by you. The LSF Administrator can terminate jobs of all users.

To terminate a job, select the job from the GUI and then choose **File | Terminate**. This same can be done by clicking on job termination toolbar after selecting the job.

`bkill` is the command line interface for terminating a job. For example,

```
% bkill 3467
Job <3467> is being terminated
```

## Terminating a Group of Jobs

You can perform a termination operation for all jobs inside a job group tree by choosing **File | Terminate Jobs**, or by running the `bkill` command with option `-R` and by specifying job group path. For example,

```
% bkill -R -J /risk_group/
Job <1413> is being terminated
Job <1414> is being terminated
Job <1415> is being terminated
Job <1416> is being terminated
```

If you only want to kill all jobs at one group level, do not use `-R`,

```
% bkill -J /risk_group/
Job <1413> is being terminated
Job <1414> is being terminated
```

## Sending Arbitrary Signals to Jobs

You can use `bkill` to send an arbitrary signal to your job using the `-s` option. You can specify either the signal name or the signal number. On most versions of UNIX, signal names and numbers are listed in the `kill(1)` or `signal(2)` manual page. On Windows NT, only customized applications will be able to process job control messages specified with the `-s` option.

For example,

```
% bkill -s SIGTSTP 3488
Job <3488> is being signalled
```

This example sends the `SIGTSTP` signal (terminal stop) to the job.

You can also use `bkill` to send signals to all jobs inside a group tree or at one group level by using `-R` option and by specifying the group path.

## Suspending and Resuming Jobs

Suspending a running job stops the job from running, freeing up CPU and memory resources. However, the job is not killed. It is still kept in the virtual memory and can be resumed later. On UNIX, this is implemented by sending a STOP signal to the job. On NT, an equivalent function is implemented.

Suspending a job that has not started yet causes the job to go into PSUSP status. A job in PSUSP status is held from scheduling until it is released explicitly.

To suspend a job, select it and then choose **Job | Suspend**. You can resume a job in the same way.

There are also command line tools for suspending and resuming jobs. To suspend a job, run the `bstop` command:

```
% bstop -J diskcheck
Job <7848> is being stopped
```

Use the `bresume` command to resume it:

```
% bresume -J diskcheck
Job <7848> is being resumed
```

Note that the `-J` option of all commands refers to a job name. If the job name does not start with a "/", it is supposed to be in the Root job group ("/"). If a job name ends with a "/", then it is a pure job group path.

Resuming a user-suspended job does not immediately put your job into RUN state. The job must first satisfy its dependency conditions. bresume first puts your job into SSUSP state. The job can then be scheduled accordingly.

You can do group wide job suspending and resuming by specifying a group path and by using the -R option of these commands. This is similar to all other job control commands we have discussed in the previous sections.

### Forcing a Job to Run

A job can be forced to run regardless of its scheduling conditions. This is desirable in certain situations, e.g., when performing a corrective action as a result of an unscheduled job.

You can force your own jobs as well as jobs submitted by other users under a job group owned by you. The LSF administrator can force any users' jobs to run.

To force a job to run, select the job and choose **File | Run Now**. The command line equivalent of this function is the brun command. For example,

```
% brun -m hosta 7884
Job <7884> is being forced to run.
```

Note that you must specify a host name on which the job should run.

## Job Group Control

Job groups are containers for jobs. As such, we can control the schedule of the whole group of jobs at the group level. Job group control can change the status of a job group into HOLD or release its status from HOLD.

Putting a job group on hold prevents jobs under the group from being scheduled. There are several situations where holding a job group is useful. If you want to define many jobs under a job group tree, while you are in the process of defining inter-job dependencies, you do not want LSF JobScheduler to schedule any of the jobs because you have not finished defining all the necessary dependencies. If you want to make

modifications to several jobs in a job group tree, you do not want the jobs to be accidentally scheduled before you finish the changes.

By default, when a new job group is created, it is put into HOLD status. You must explicitly release the job group by releasing it. To release a job group, select the job group and choose **File | Release**. When you hold or release a job group from the GUI, all sub-groups will also be held or released recursively.

If you want to hold a job group, select the job group and choose **File | Hold**.

The tools '`bghold`' and '`bgrel`' can be used to perform the equivalent operations from the command-line. Both commands take a group name as a parameter. For example:

% **bghold /a/b/c**

will hold the group /a/b/c and all subgroups.

% **bghold "/a/b/c/g*"**

will hold the groups beginning with 'g' under the group /a/b/c.

% **bgrel /a/b/c**

will release the group /a/b/c and all subgroups.

% **bgrel -d /a/b/c**

will release the group /a/b/c but not any sub-groups.

# Managing Schedules of Jobs

Sometimes you may need to examine the schedules of your jobs to make sure the schedules are met. You may have made a mistake in defining jobs and thus the jobs were not scheduled as expected. Or certain events had not happened for some reasons.

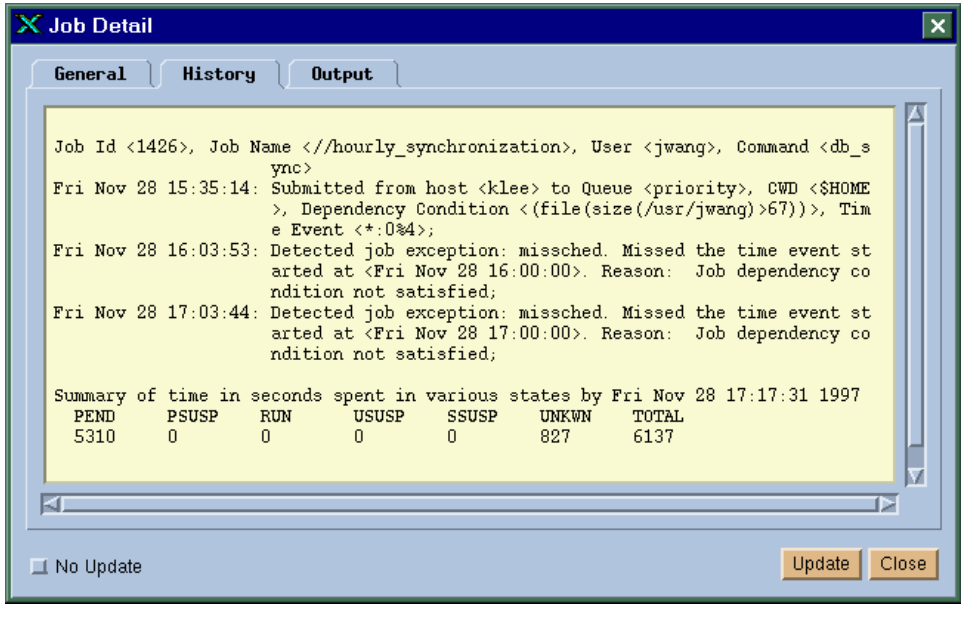
There can be many reasons causing the job to not run. One obvious reason might be that you forgot to release the job group so that it is no longer in HOLD status.

You may have specified a resource requirement that will never be satisfied, or the system load has been too heavy to be able to run any additional jobs. You may have specified a duration for a time event that is too short for a host to become available to run your job.

Fortunately, LSF JobScheduler provides you with the information you would need to find out why your jobs did not make the schedule.

*Figure 26* shows a job history window that explains what happened to your job's schedule.

Figure 26. Job Schedule History




# System Status Monitoring

Frequently you will need to know what is happening in the system and relate it to your job's schedule, and take some actions to correct problems with your network. LSF

JobScheduler provides you with a complete picture of what is happening to your system from different angles.

## Event View

If your job depends on one or more events, you should check the status of the events so that you know what has caused your job to run or not to run unexpectedly.

The status of all job or job group related events is obvious from the job or job group details, as was discussed in *'Viewing Details of a Job or Job Group' on page 95*.

The status of time events can be viewed by looking at the date and time parameters of the job and by checking the status of the calendar. The status of calendars can be viewed using `xbcal` GUI, as was discussed in *'Using the LSF JobScheduler - Calendar GUI' on page 33*. You can also view calendar status using the `bcal` command, as was discussed in *'Manipulating Calendars Using the Command Line Interface' on page 40*.

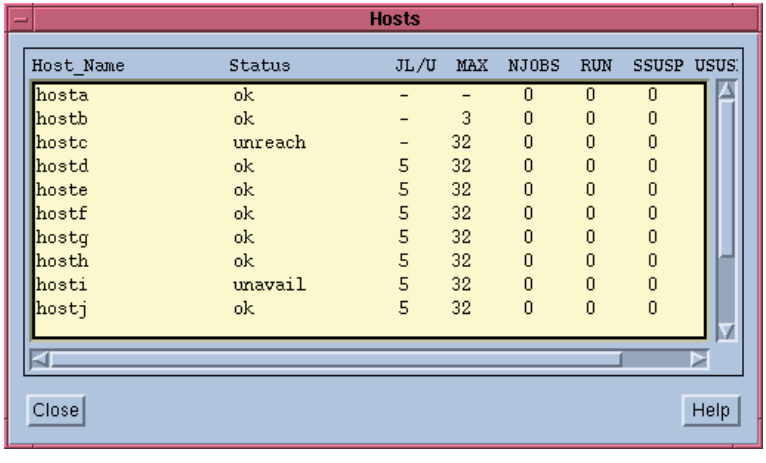
For file events, exception events, and user events, you can view their status by running the `bevents` command, as was discussed in *'Viewing Events' on page 29*.

## Host View

To know the status of each LSF JobScheduler server host, choose **Tools | Hosts** in the `xlsjs` main window. This command displays the host window shown in *Figure 27*. Right-click to customize the view. The status of hosts is displayed in the List view, and is indicated by colour in the icon view.

Figure 27. Host View of LSF JobScheduler

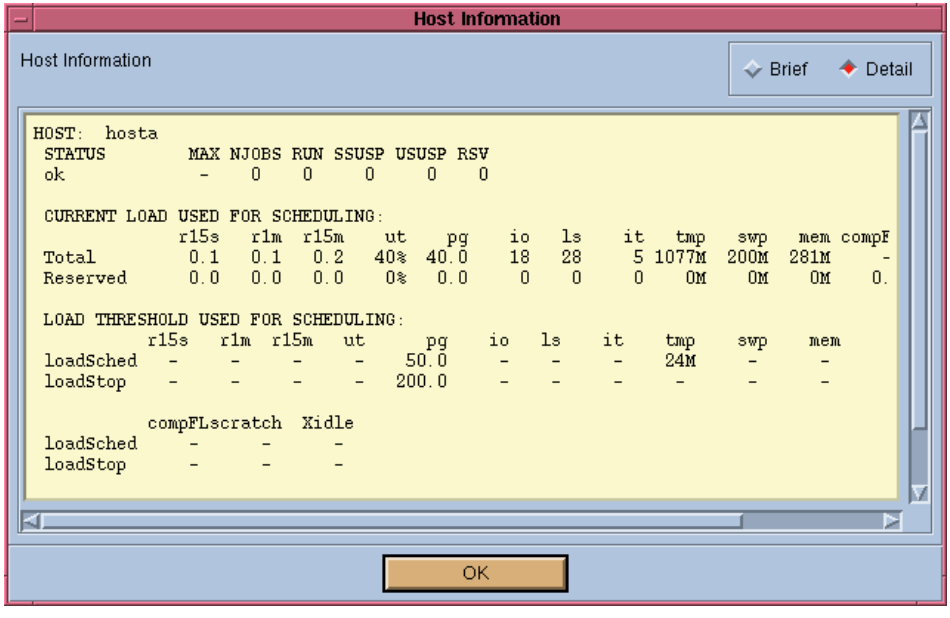
Figure 28. Host View in List Format




| Host_Name | Status | JL/U | MAX | NJOBS | RUN | SSUSP | USUS |
|---|---|---|---|---|---|---|---|
| hosta | ok | - | - | 0 | 0 | 0 | |
| hostb | ok | - | 3 | 0 | 0 | 0 | |
| hostc | unreach | - | 32 | 0 | 0 | 0 | |
| hostd | ok | 5 | 32 | 0 | 0 | 0 | |
| hoste | ok | 5 | 32 | 0 | 0 | 0 | |
| hostf | ok | 5 | 32 | 0 | 0 | 0 | |
| hostg | ok | 5 | 32 | 0 | 0 | 0 | |
| hosth | ok | 5 | 32 | 0 | 0 | 0 | |
| hosti | unavail | 5 | 32 | 0 | 0 | 0 | |
| hostj | ok | 5 | 32 | 0 | 0 | 0 | |

Double-click any host to bring up the detailed host information window shown in *Figure 29*. This window shows all the configuration information of the host and its current status—such as how many jobs are currently running on the host, and recent load information used for controlling jobs. The “Reserved” line shows the load reserved for jobs that reserved resources. See *Section 4, ‘Resources’, beginning on page 45*, for more information on resource reservation.

Figure 29. Detailed Host Information

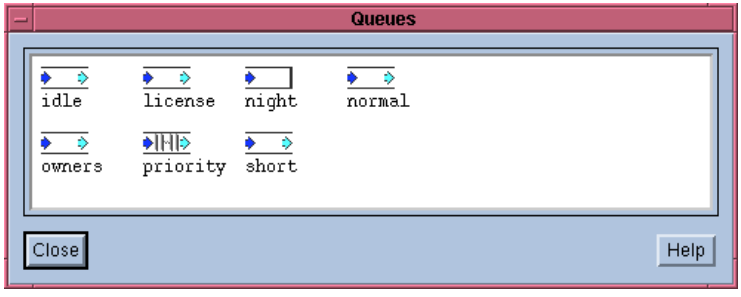



You can also view host information by running the `bhosts` command from the command line. For detailed host information, run `bhosts` with option `-l`.

## Queue View

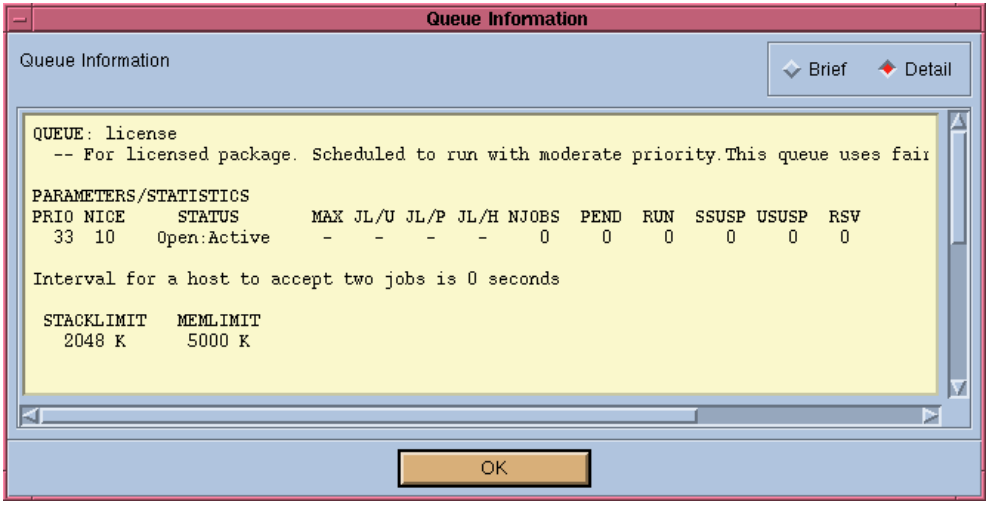
You may also want to know how many jobs are in each queue. To display the queue window shown in *Figure 30*, choose **Tools | Queues** in the `xlsjs` main window. Right-click to customize the view. The status of queues is displayed in the List view, and is indicated by different symbols in the icon view. You can also see if the queue is empty by looking at the queue box.

Figure 30. Queue Window




Double-click a queue to display detailed configuration information for the queue and statistics about jobs currently in the queue, as shown in *Figure 31*.

Figure 31. Detailed Queue Information Window

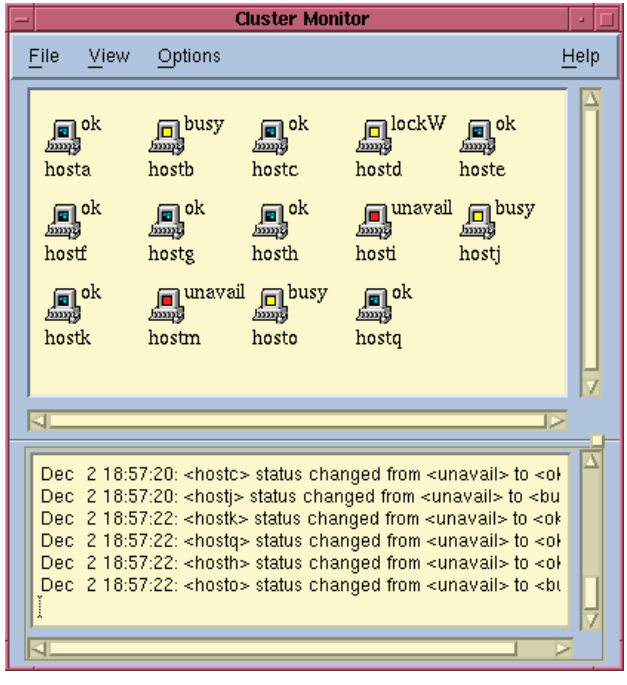



You can also view queue information using command line interface via the `bqueues` command. For detailed queue information, run `bqueues` with the `-l` option.

## Load View

You may also want to check the load situation to get an understanding of the real-time load situation on each host. The `xlsmon` GUI gives you the overall load information. *Figure 32* shows the main window of the `xlsmon` GUI which shows an icon for each host in the cluster. Each host is labelled with its status. Hosts change colour as their status changes.

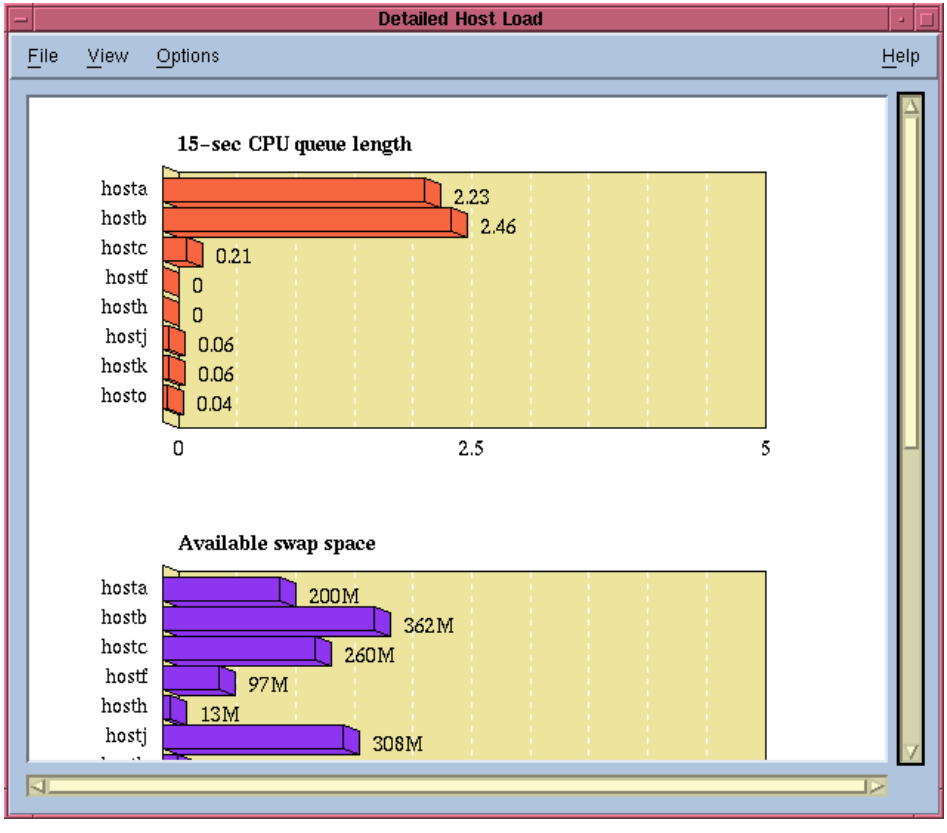
Figure 32. xlsmon GUI Main Window




You can choose other displays by selecting them from the View menu.

To view the detailed load on each machine, choose **View | Detailed Load**. This brings up a bar chart window as shown in *Figure 33*, giving load indices for each host.

Figure 33. Detailed Host Load Window



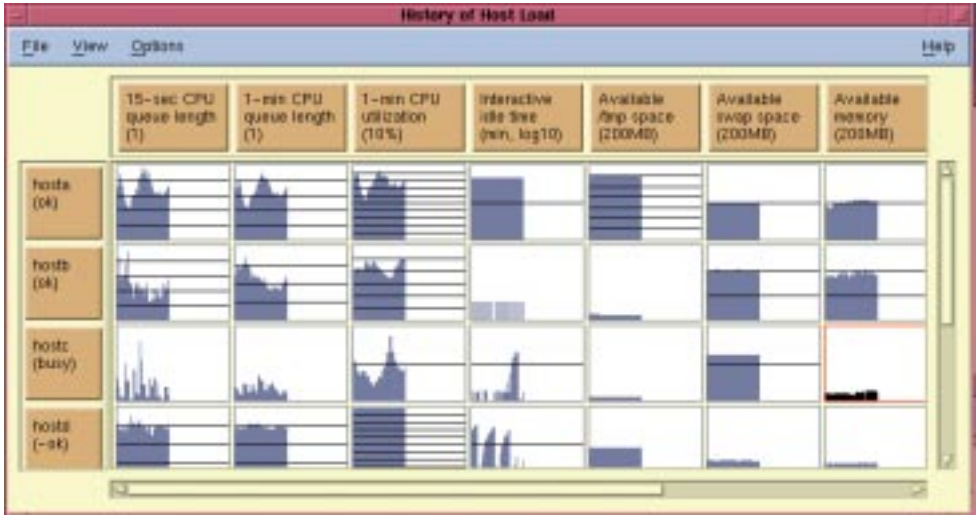

You can select which load indices and which hosts to display by choosing options from the View menu.

The same information can be displayed in text format by running the `lsload` command, or continuously monitored and displayed by running `lsmon` command.

You can customize the view of the detailed load window by selecting options from the View menu.

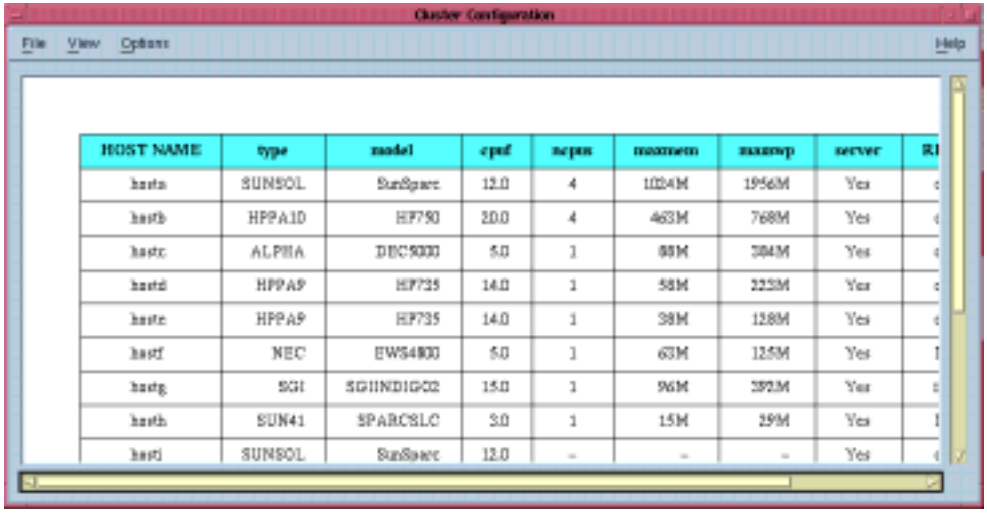
`xlsmon` also allows you to view host load history over time, as shown in *Figure 34*. As with the detailed host load window, you can select which hosts and which load indices to display by choosing options from the View menu.

Figure 34. Load History Window




Sometimes you may want to know your cluster configuration information, e.g., number of CPUs available on a machine. `xlsmon` provides a configuration view that gives you this information, as shown in *Figure 35*. This window can be accessed by choosing **View | Cluster Configuration**. This same information can be displayed using the `lshosts` command.

Figure 35. Configuration Information Window




| HOST NAME | type | model | cpuf | ncpus | maxmem | maxswp | server | RI |
|---|---|---|---|---|---|---|---|---|
| hosta | SUNSOL | SunSparc | 12.0 | 4 | 1024M | 1956M | Yes | c |
| hostb | HPPA10 | HP750 | 20.0 | 4 | 463M | 768M | Yes | c |
| hostc | ALPHA | DEC5000 | 5.0 | 1 | 88M | 384M | Yes | c |
| hostd | HPPA9 | HP735 | 14.0 | 1 | 58M | 223M | Yes | c |
| hoste | HPPA9 | HP735 | 14.0 | 1 | 38M | 128M | Yes | c |
| hostf | NEC | EWS4800 | 5.0 | 1 | 63M | 125M | Yes | I |
| hostg | SGI | SGIINDIGO2 | 15.0 | 1 | 96M | 192M | Yes | s |
| hosth | SUN41 | SPARCSLC | 3.0 | 1 | 15M | 29M | Yes | I |
| hosti | SUNSOL | SunSparc | 12.0 | - | - | - | Yes | c |

# 7. Exception Handling and Alarms

When managing critical jobs it is important to ensure that the jobs run properly. When problems arise with the scheduling and execution of a job, it is necessary that some actions be taken to fix the problem. An alarm specifies how a notification should be sent in the event of such a problem.

LSF JobScheduler provides flexible ways to handle failures and exceptions so that you can define what to do when certain exceptions happens. Failures of job processing are defined in terms of exceptions and handling of these failures are defined in terms of exception handlers. An alarm is triggered as a result of the exception handler `alarm()`.

## Exceptions

LSF JobScheduler monitors exceptions as defined by exception functions. Each exception function has a name and possibly parameters that allow users to further customize exceptions on a per job basis.

The exception functions are used when creating a job to tell LSF JobScheduler what exceptions to watch for for this particular job. You need to understand the behaviour of your application properly to determine what is considered an exception for your job. The following are the exception functions supported by LSF JobScheduler:

**`missched()`**

The job has a time event associated with it, and the job was not able to start while the time event was in active status. This function has no parameter associated with it.

There can be many different reasons why your job can miss its schedule. For example, you may have specified a resource requirement for your job that was not satisfied while the time event was active. Or the duration of the time event is too short to find a host that can process the job. It is also possible that you have specified other dependency conditions for the job that were not satisfied while the time event was active.

To minimize such exceptions, you should carefully examine your schedule and conditions, and make sure that in most cases the conditions will be met when the time event is active.

**`abend(ec1, ec2, ...)`**

The job has exited with one of the given exit codes. The list of parameters is a list of exit codes in the range of -128 to 127. Each parameter can be either one exit code, or a range of codes in the form of c1-c2.

Most applications, when finishing successfully, should exit with exit code 0. However, some applications are not programmed to handle exit codes properly. For example, some applications exit with a non-zero value even if it finishes successfully. You should carefully check your application and determine the behaviour of your job, and define your own job-specific abnormal termination accordingly. Sometimes you may have to wrap your application with a script to make its exit code reflect success or failure.

**`overrun(max_time)`**

The job has run for too long. The parameter specifies the maximum allowed run time in minutes. This function can be used to detect a situation where a job runs away, or when a job hangs. For example, if your job should finish in less than 10 minutes, then if the job has run for 2 hours, something must be wrong.

**`underrun(min_time)`**

The job has finished too soon. The parameter specifies the minimum required run time in minutes. This function is used to detect situations where a job finishes prematurely.

**`startfail()`**

A problem in starting the job has occurred and thus the job was unable to start. This function has no parameter.

Typical reasons for this exception include lack of system resources, e.g. process table full on the execution host, or file system not mounted properly thus the execution host cannot set up the execution environment for the job.

**hostfail()**

The host on which a job was started went down. This function has no parameter.

**cantrun()**

System detects that it is impossible to run the job because various dependencies cannot be satisfied. This exception also happens if `startfail()` exception occurs 20 times in a row.

A typical example of a job that triggers `cantrun()` exception is a job that depends on an job event, but that job has been deleted from the system so that the job event never happens.

# Exception Handlers

For each exception condition of a job, a corrective action can be associated with the job which is automatically invoked when the specified exception happens. Such an action is called an exception handler in LSF JobScheduler. The handler can try to resolve the problem automatically or inform the user/administrator that the problem was detected. The currently available handlers are:

**rerun()**

Rerun the job. In many situations, re-running the job will fix the problem. This handler is only relevant for exception conditions `underrun()` and `abend()`. It is possible that the job will be re-run on a different host depending on the dynamic resource availability and the job's resource requirement.

**alarm(severity, alarm_name)**

Raise an alarm with the given severity. The alarm name identifies which alarm should be triggered. The valid alarm names are configured by your LSF administrator and can be viewed by `balarms` command. See *'Alarms' on page 130* for details of alarms.

**kill()**

- This action causes the current execution of the job to be terminated. This action can only be specified for the `overrun()` exception.

**`setexcept(event_name)`**

Set the exception event identified by `event_name` to active. The parameter `event_name` is the name of an event that is created by another job as an exception event dependency condition. See *'Job Exception Event Dependency' on page 89* for details.

This handler is an interface for external exception handlers. All other exception handlers are built-in handlers. By defining jobs that respond to the job exception event, arbitrary actions can be invoked to handle the exception. This is useful when none of the built-in handlers can handle your particular error.

# Using Exception Handlers

With the exception handling mechanism provided in LSF JobScheduler, you can tune your schedules so that all failures are taken care of automatically, and minimum human intervention is necessary. This section goes through some typical steps of implementing error handling measures for your schedules.

## Handling Failures with Built-in Exception Handlers

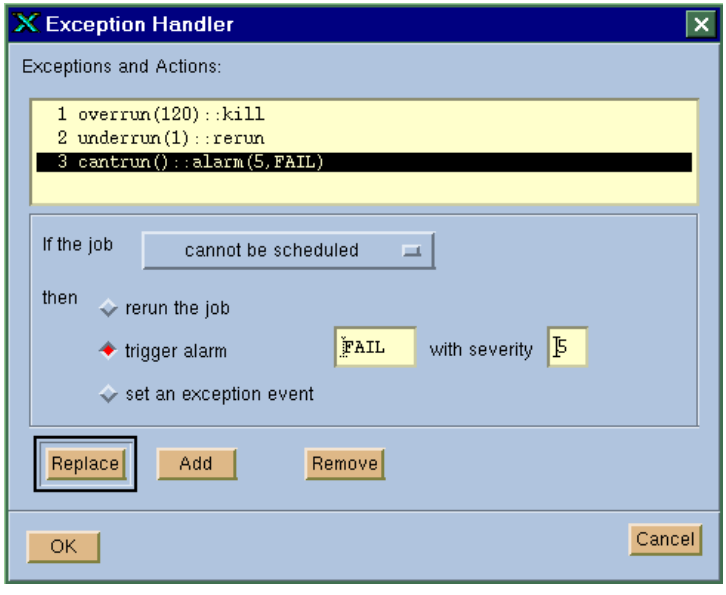
When you create a job, you can specify what exceptions you want LSF JobScheduler to watch for, and how you want to handle the exceptions when they happen. Click on the "Exception Handler" button in the job submission window, as shown in *Figure 11 on page 64*, to display the exception handler window, as shown in *Figure 36*.

You can specify any number of exceptions, each with a handler.

In the example in *Figure 36*, three exceptions for the job have been defined, each with a different built-in exception handler. The first exception is `overrun(120)` which tells LSF JobScheduler that this job is not to run for more than 120 minutes. If this happens, the job should be killed, as indicated by the handler `kill()`. The second exception is `underrun(1)` which tells LSF JobScheduler that this job should not run for less than 1 minute. If this happens, the job should be re-run, as specified by the handler `rerun()`. The last exception is `cantrun()`, which tells LSF JobScheduler to raise the alarm named "FAIL" with severity 5 if the job is impossible to schedule.

For details on raising an alarm handler, see *'Alarms' on page 130*.

Figure 36. Exception Handler Definition Window



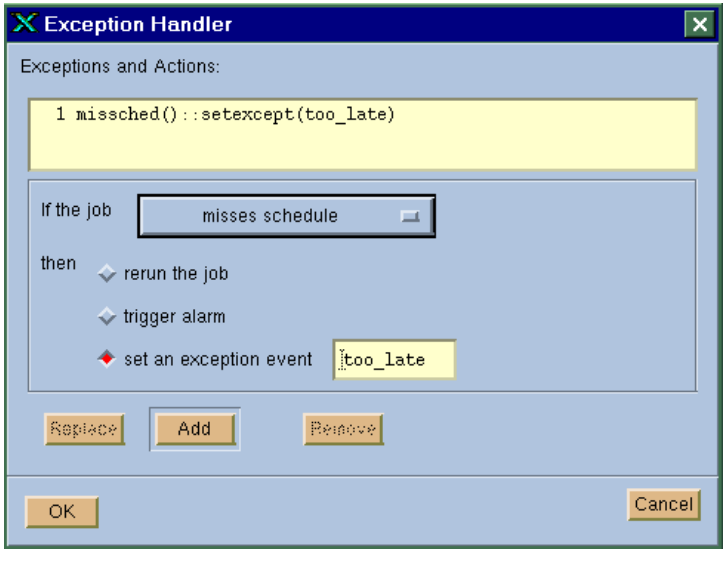

## Handling Failures with Recovery Jobs

If none of the built-in handlers serve your purpose, you can set up your job so that external exception handlers or recovery jobs can be automatically invoked to correct the error. In order to achieve this, an exception event must be defined together with the recovery job. Multiple error recovery jobs can be defined to respond to the same job exception event.

First, you define the main job using the job submission window and define "`setexcept()`" to be the exception handler, as shown in *Figure 37*. By defining the exception handler as `setexcept()`, the job exception event "`too_late`" is set to active status as soon as the exception "`missched()`" is detected for the current job.

The status change of event "`too_late`" can then trigger all recovery jobs that depend on the "`too_late`" job exception event.

Figure 37. Setting an Exception Event to Trigger Recovery Jobs



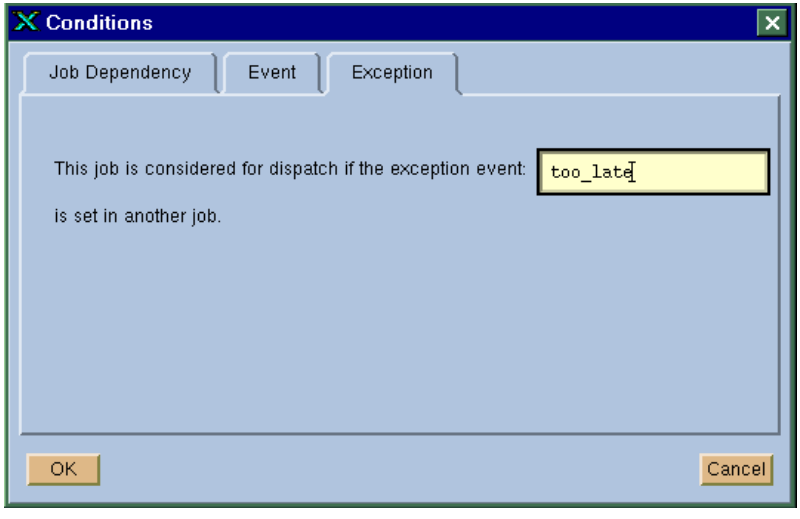

To define an error recovery job that responds to the job exception event, click on the "Conditions" button in the job submission window as shown in *Figure 11 on page 64*, and then click on the "Exception" tab. This brings up the exception condition window as shown in *Figure 38*. Enter the name of the job exception event in the window to make this job respond to the event.

As soon as you define a job that depends on the named exception event, the event is created inside LSF JobScheduler which then starts to monitor the status of the event. Whenever the exception happens, the event will be set to ACTIVE as a result of the `setexcept()` handler of the main job, thus triggering the recovery job to run.

Note that you can define more than one recovery job to respond to the same exception. On the other hand, you can also have one recovery job to be the handler for many main jobs, as long as they all trigger the same exception event.

It is possible to set your job such that some exceptions are handled by built-in exception handlers while others handled by recovery jobs. As shown in *Figure 36*, you can have a different exception handler for each exception condition.

Figure 38. Setting a Job to Respond to an Exception Event




## Setting Exception Handlers Using Command Line Interface

Although you can use the graphical tools for all exception handling, LSF JobScheduler also allows you to define exception handling from the command line using the `bsub` command.

To specify an exception handler for a job, use the "`-X`" option of the bsub command. The format of the -X option is:

*`exception_cond([params])::action`*

where `exception_cond` is one of the exception functions discussed in *'Exceptions' on page 123*, `params` are possible parameters associated with the exception function, and "`action`" is one of the exception handlers discussed in *'Exception Handlers' on page 125*.

The following example sets an exception handler for a job which will result in an alarm being triggered if the job exits with an exit code of 10:

```
% bsub -X “abend(10)::alarm(5, pageadmin)” [other options] myjob
```

Multiple handlers can be specified by repeating the -X option:

```
% bsub -X “abend(10)::alarm(5, pageadmin)” -X /
      “abend(1)::rerun” [other options] myjob
```

To define a job that responds to a job exception event, use the `-w` option of the `bsub` command. The following example defines two jobs. The first job is the main job and the second job is a recovery job that is triggered when the first job has failed to finish within 60 minutes after starting.

```
% bsub -X “overrun(60)::setexcept(too_long)” [other options] realJob
% bsub -w “exception(too_long)” [other options] recoveryJob
```

# Alarms

An alarm specifies how a notification should be sent in the event of an exception. An alarm is triggered as a result of the exception handler `alarm()`. If you want to trigger an alarm as a result of a job failure, you must specify an alarm name and a severity number using the `alarm()` exception handler.

## How Are Alarms Generated

When the `alarm()` handler is called, LSF JobScheduler invokes a site-installable executable that must be named `raisealarm`. This executable is a reporter which raises the specified alarm. LSF JobScheduler invokes the `raisealarm` executable with four arguments: alarm name, severity, source, and context. The source argument is the

name of the job that has triggered the alarm. The context tells the detail of the alarm, for example,

```
overrun Job[1529] User[user1] Queue[normal]
```

LSF JobScheduler comes with a default `raisealarm` executable that will do a few things when an alarm is triggered. It first logs a record for the alarm incident in a log file. It then initiates a notification method defined in the alarm configuration. LSF JobScheduler also re-sends the notification if an open alarm has not been acknowledged within a given time.

The `raisealarm` executable bundled with LSF JobScheduler can be configured to send a notification via email, or invoke a site-replaceable notification executable.

If your site is using the `raisealarm` executable bundled with LSF JobScheduler, your LSF administrator can define alarms through a configuration file. To see what alarms are currently configured by your LSF administrator, run the `balarms` command with the `def` option. For example:

```
% balarms def
ALARM: pageadmin
-- Page the administrator

NOTIFICATION:
METHOD:  CMD[ /usr/local/bin/pageadmin]
RETRIES: Every 30 minutes. Maximum of 10 retries

EXPIRATION:
Incident automatically expires after 6000 minutes

---------------------------------------------------

ALARM: MailAdmin
-- Inform admin of minor problems

NOTIFICATION:
METHOD:  EMAIL[admin]
RETRIES: Every 30minutes. Maximum 5 retries
```

```
EXPIRATION:
Incident automatically expires after 3000 minutes
```

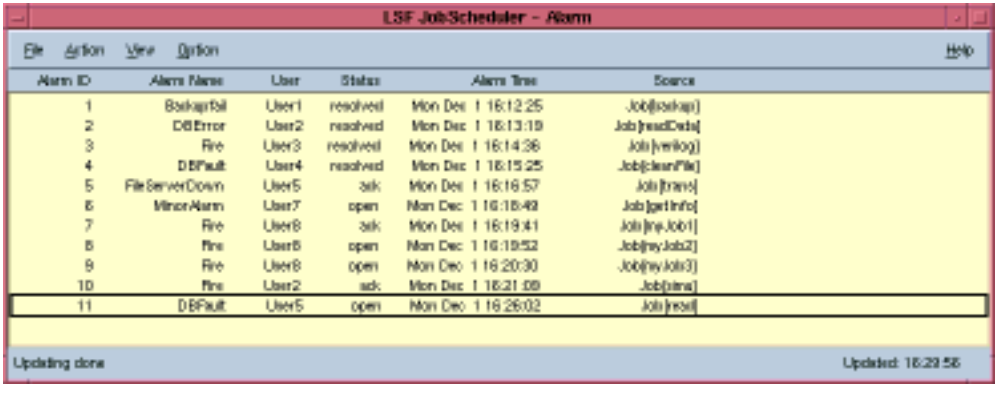
Each alarm has a name and a notification method defined. LSF JobScheduler supports notification through email or using a configured command which can be used to send a notification via a pager, consoles, etc. The administrator can configure an alarm to periodically resend the notification if the alarm is not acknowledged. You can also view alarm configuration from the `xbalarms` GUI.

## Manipulating Alarms

Each time an alarm is triggered, a new alarm record is added to a log file. The alarm record contains the name of the alarm, when it was triggered, the job that has triggered the alarm, and some additional details.

You can view and manipulate alarms by using the `balarms` command or the `xbalarms` GUI. *Figure 39* shows a sample listing of outstanding alarm incidents using the `xbalarms` GUI.

Figure 39. Alarm Monitoring Window




The possible states of an alarm are:

**open**
: The alarm is triggered, but has not been acknowledged.

**ack**

The alarm is acknowledged by some user, which should indicate that the user has processed the alarm situation.

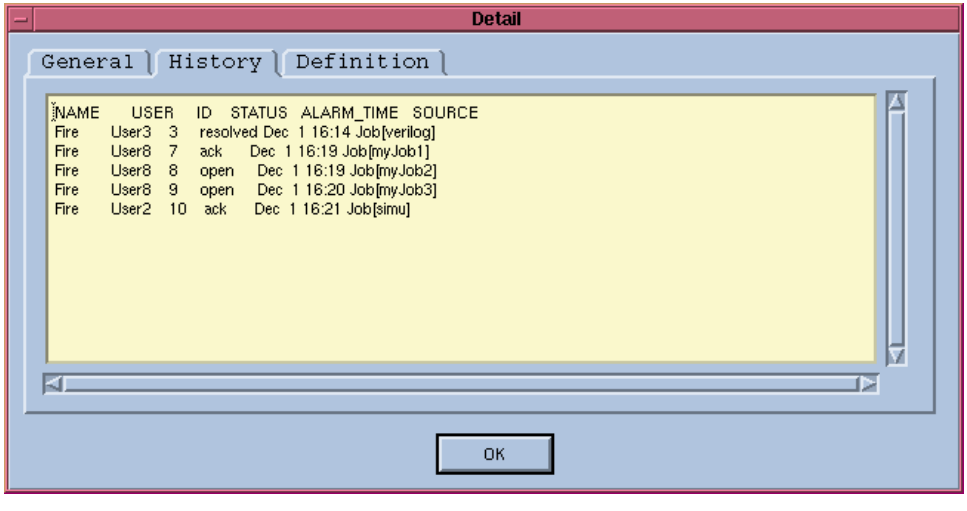
**resolved**

The alarm situation has been resolved. This means that this alarm has been handled and no longer needs attention.

You can view the details of each alarm by double-clicking on the selected alarm. This will bring up a popup window with alarm details, which allows you to see alarm configuration, alarm status, and alarm history. *Figure 40* is an example of the alarm history window of the `xlsbalarms` GUI.

To acknowledge or resolve an alarm, select the alarm and choose the appropriate action from the "Action" menu. This will bring up a window prompting you for a resolving or acknowledging message. The message you enter will be logged so that other people will be able to see your solutions in the alarm detail window.

Figure 40. Alarm History Window




The above alarm system interface is valid if your site is using the bundled `raisealarm` executable of LSF JobScheduler. Your site may choose to handle alarms generated by JobScheduler jobs through another alarm management module which

may interpret the severity field differently. In that case, you should refer to local documentation for alarm management.

# Index

## A


address (Platform) . . . . . xi
alarm reporter . . . . . 130
Alarms . . . . . 130
aliases for resource names. . . . . 52


## B


batch jobs
    resource reservation . . . . . 55
bcadd . . . . . 12
`bcadd` . . . . . 12
`bcal` . . . . . 14, 83
`bdel` . . . . . 16, 107
bgadd . . . . . 77
`bkill` . . . . . 110
`bmodify` . . . . . 105
boolean resources . . . . . 51
bparams . . . . . 70
`bparams` . . . . . 70
`bresume` . . . . . 111
`bstop` . . . . . 111
`bsub` . . . . . 15, 63
built-in event . . . . . 21
built-in exception handler . . . . . 126


## C


calendars . . . . . 5
    adding . . . . . 12
    associating jobs . . . . . 15
    displaying . . . . . 14
    removing jobs . . . . . 16
contacting Platform Computing . . . . . xi
CPU factor . . . . . 50
`cpuf` static resource . . . . . 50
current job group. . . . . 78


## D


default queue . . . . . 69
default resource requirement . . . . . 71
dependency keyword
    `done` . . . . . 85
    `ended` . . . . . 85
    `exit` . . . . . 85
    `file` . . . . . 86
documentation. . . . . x


## E


effective run queue length . . . . . 49
ELIM (External Load Information
        Manager) . . . . . 47
environment variable
    `LSB_DEFAULTQUEUE` . . . . . 70
events . . . . . 4
    built-in event. . . . . 21
    external event . . . . . 27
    file event . . . . . 27
    prior job event . . . . . 23
    status . . . . . 22
    user event . . . . . 28
exception. . . . . 123
exception function. . . . . 123
exception handler . . . . . 125
execution priority . . . . . 50
external event. . . . . 27

external exception handler . . . . . 126, 127
external load index . . . . . 47

## F

fax numbers (Platform) . . . . . xi
file event . . . . . 27
file event function
    age . . . . . 28
    arrival . . . . . 27, 88
    exist . . . . . 27, 88
    size . . . . . 28, 88
file transfer . . . . . 73

## G

guides . . . . . x

## H

help . . . . . x, xi
hname static resources . . . . . 50
host group . . . . . 68
host preference . . . . . 69
host selection, *see* resource requirements
host status
    lockW . . . . . 49
host types . . . . . 50

## I

input and output . . . . . 65

## J

job control messages . . . . . 109
job control signals . . . . . 109
job group . . . . . 75, 76
job group path . . . . . 60, 75
job spanning, *see* resource requirements
job state
    PSUSP . . . . . 63
    SSUSP . . . . . 63
    USUSP . . . . . 63
Job Status . . . . . 60
jobs . . . . . 4
    associating with calendars . . . . . 15
    attributes
        jobID . . . . . 60
        jobName . . . . . 60
    execution history . . . . . 99
    input and output . . . . . 65
    inter-job dependencies . . . . . 83
    modification . . . . . 104
    numeric job names . . . . . 86
    pre-execution commands . . . . . 72
    removing from calendars . . . 16, 107
    submitting . . . . . 63
    suspending and resuming . . . . . 111

## L

load index
    r15m . . . . . 49
    r15s . . . . . 49
    r1m . . . . . 49
    swp . . . . . 49
    tmp . . . . . 49
load indices
    built-in . . . . . 47
    external . . . . . 47
lockW host status . . . . . 49
LSF Enterprise Edition . . . . . x
LSF Standard Edition . . . . . x
LSF Suite documentation . . . . . x
LSF Suite products . . . . . ix
lsf.task file . . . . . 57
lsftask . . . . . 57

lsinfo ..... 47
lsload
  -E option ..... 49
  -N option ..... 49
lsrtasks ..... 57

## M

mailing address (Platform) ..... xi
maxmem static resource ..... 50
maxswp static resource ..... 50
maxtmp static resource ..... 50
messages, job control ..... 109
model static resource ..... 50

## N

ncpus static resource ..... 50
ndisks static resource ..... 50
normalized run queue length ..... 49

## O

online documentation ..... xi
order string, *see* resource requirements

## P

parallel jobs
  locality ..... 55
phone numbers (Platform) ..... xi
Platform Computing Corporation .... xi
pre-execution command ..... 72
pre-execution commands ..... 72
prior job event ..... 23
PSUSP job state ..... 63

## Q

queues ..... 5
  choosing ..... 70

## R

r15m load index ..... 49
r15s load index ..... 49
r1m load index ..... 49
raisealarm ..... 130
remote jobs
  execution priority ..... 50
remote task list, *see* task lists
resource name aliases ..... 52
resource requirement ..... 71
resource requirements
  format ..... 51
  ordering hosts ..... 52, 54
  parallel job locality ..... 55
  resource reservation ..... 55
  resource usage ..... 52, 55
  selecting hosts ..... 52
resource usage, *see* resource requirements
resources
  boolean ..... 51
  static ..... 50
rexpri static resource ..... 50
run queue
  effective ..... 49
  normalized ..... 49
rusage, *see* resource requirements

## S

selection string, *see* resource requirements
server static resource ..... 50

signals, job control . . . . . 109
SSUSP job state . . . . . 63
static resource
    hname . . . . . 50
static resources . . . . . 49, 50
    cpuf . . . . . 50
    maxmem . . . . . 50
    maxswp . . . . . 50
    maxtmp . . . . . 50
    model . . . . . 50
    ncpus . . . . . 50
    ndisks . . . . . 50
    rexpri . . . . . 50
    server . . . . . 50
    type . . . . . 50
support . . . . . xi
swp load index . . . . . 49

## T

task lists . . . . . 57
    files . . . . . 57
    remote . . . . . 57
technical assistance . . . . . xi
telephone numbers (Platform) . . . . . xi
tmp load index . . . . . 49
type static resource . . . . . 50

## U

user event . . . . . 28
USUSP job state . . . . . 63

## W

working group . . . . . 79

## X

xbcal . . . . . 13
xbsub . . . . . 16