Platform LSF Programmer's Guide

Platform LSF
Version 7.0 Update 6
Release date: August 2009
Last modified: August 17, 2009

We’d like to hear from you

You can help us make this document 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 document, please address your comments to doc@platform.com.

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

Contents

CHAPTER

1

Introduction

Platform LSF architecture

Platform LSF is a layer of software services on top of UNIX and Windows operating systems. Platform LSF creates a single system image on a network of different computer systems so all the computing resources on a network can be managed and used. Throughout the LSF Programmer's Guide, Platform LSF refers to the Platform LSF suite, which contains the following products:

LSF base

LSF base provides basic load-sharing services to a network of different computer systems. All LSF products use LSF base. Some of the services it provides are:

- Resource information
- Host selection
- Job placement advice
- Transparent remote execution of jobs
- Remote file option

To provide services, LSF base includes:

- Load Information Manager (LIM)
- Process Information Manager (PIM)
- Remote Execution Server (RES)
- LSF base API
- `lstools`
- `lstcsh`
- `lsmake`

LSF batch

The services provided by LSF batch are extensions of the LSF base services. LSF batch makes a computer network a network batch computer. It has all the features of a mainframe batch job processing system while doing load balancing and policy-driven resource allocation control.

LSF batch relies on services provided by LSF base. LSF batch uses:

- Resource and load information from LIM to do load balancing
- Cluster configuration information from LIM
- The master LIM election service provided by LIM
- RES for interactive batch job execution
- Remote file operation service provided by RES for file transfer

LSF batch includes a master batch daemon (`mbatchd`) running on the master host and a slave batch daemon (`sbatchd`) running on each batch server host.

LSF libraries

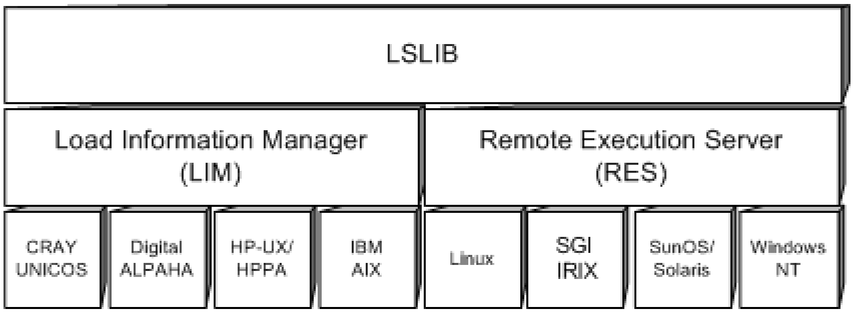
Platform LSF consists of a number of servers running as root on each participating host in an Platform LSF cluster and a comprehensive set of utilities built on top of the Platform LSF API. The Platform LSF API consist of two libraries:

- LSLIB, the Platform LSF base library, provides Platform LSF base services to applications across a heterogeneous network of computers.

- LSBLIB, the LSF batch library, provides batch services to submit, control, manipulate, and queue jobs. LSBLIB also provides access to the services of other LSF products.

LSF base system

The diagram below shows the components of the Platform LSF base and their relationship:



LSF base consists of the Platform LSF base library (LSLIB) and two servers daemons, the Load Information Manager (LIM) and the Remote Execution Server (RES).

LSLIB

The LSF API LSLIB is the direct user interface to the LSF base system. Platform LSF APIs provide easy access to the services of Platform LSF servers. An Platform LSF server host runs load-shared jobs. A LIM and a RES run on every Platform LSF server host. They interface with the host’s operating system to give users a uniform, host-independent environment.

Cluster

A cluster is a collection of hosts running LSF. A LIM on one of the hosts in a cluster acts as the master LIM for the cluster. The master LIM is chosen among all the LIMs running in the cluster based on configuration file settings. If the master LIM becomes unavailable, the LIM on the next configured host will automatically become the new master LIM.

LIM

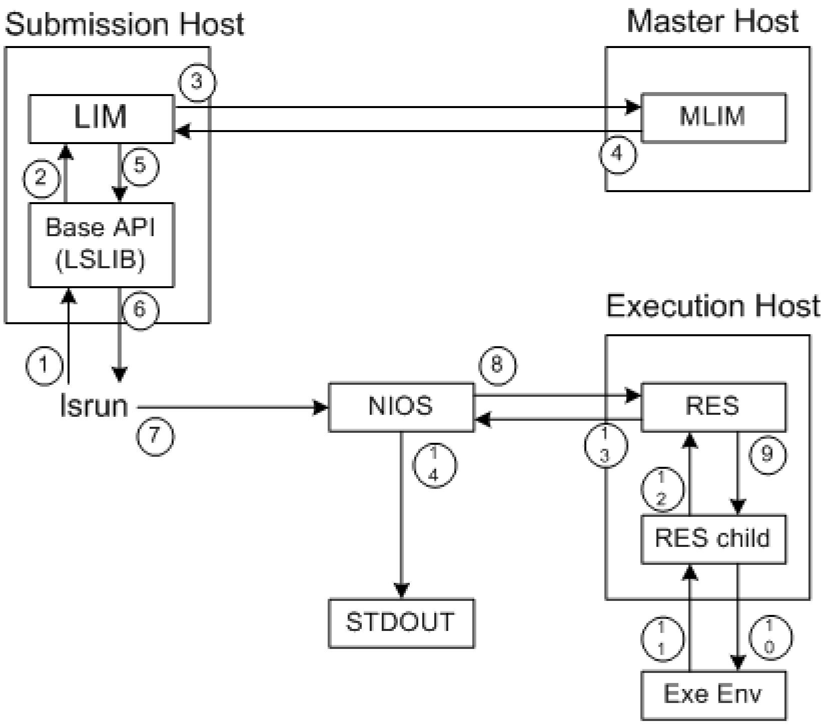
The LIM on each host monitors its host's load and reports load information to the master LIM. The master LIM collects information from all hosts and provides that information to the applications.

RES

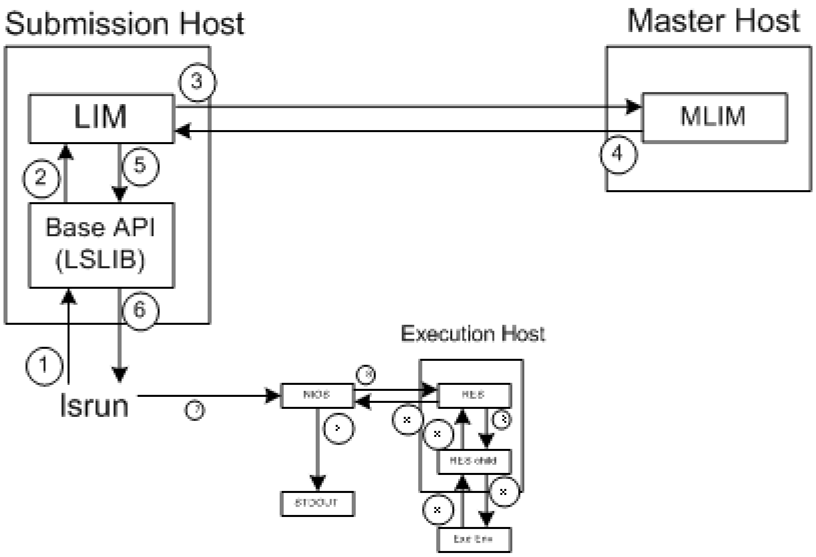
The RES on each server host accepts remote execution requests and provides fast, transparent, and secure remote execution of tasks.

Application and Platform LSF base interactions

The following diagram shows how an application interacts with Platform LSF base. All of the transactions take place transparently to the programmer:

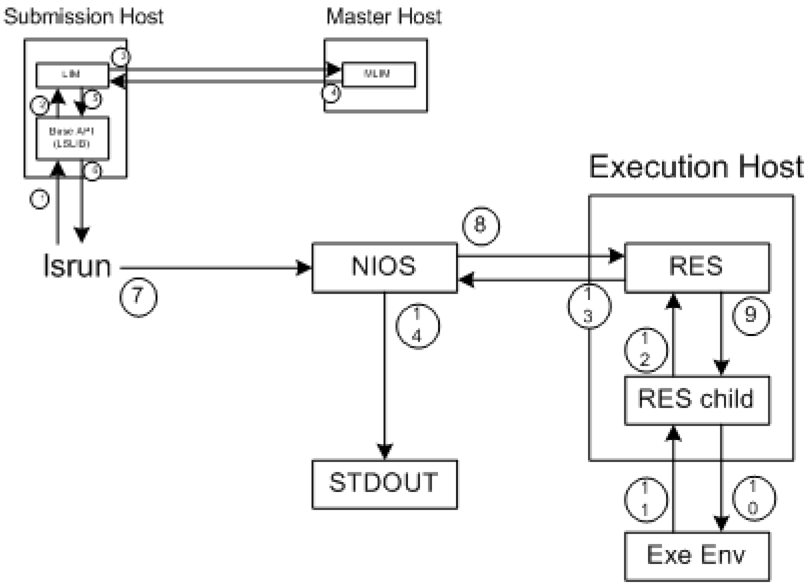


LSF base executes tasks by sending user requests between the submission, master, and execution hosts. From the submission host send a task into the LSF base system. The master host determines the best execution host to run the task. The execution host runs the task.



1. `lsrun` submits a task to LSF for execution.
2. The submitted task proceeds through the Platform LSF base library (LSLIB).
3. The LIM communicates the task's information to the cluster's master LIM. Periodically, the LIM on individual machines gathers its 12 built-in load indices and forwards this information to the master LIM.

4. The master LIM determines the best host to run the task and sends this information back to the submission host's LIM.
5. Information about the chosen execution host is passed through the LSF base library.
6. Information about the host to execute the task is passed back to `lsrun`.



7. `lsrun` creates NIOS (network input output server) which is the communication pipe that talks to the RES on the execution host.
8. Task execution information is passed from the NIOS to the RES on the execution host.
9. The RES creates a child RES and passes the task execution information to the child RES.
10. The child RES creates the execution environment and runs the task.
11. The child RES receives completed task information.
12. The child RES sends the completed task information to the RES.
13. The output is sent from the RES to the NIOS. The child RES and the execution environment is destroyed by the RES.
14. The NIOS sends the output to standard out

To run a task remotely or to perform a file operation remotely, an application calls the remote execution or remote file operation service functions in LSLIB, which then contact the RES to get the services.

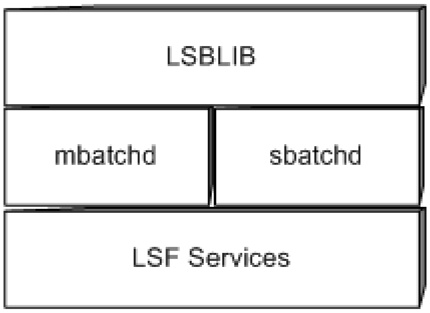
The same NIOS is shared by all remote tasks running on different hosts started by the same instance of LSLIB. The LSLIB contacts multiple Remote Execution Servers (RES) and they all call back to the same NIOS. The sharing of the NIOS is restricted to within the same application.

Remotely executed tasks behave as if they were executing locally. The local execution environment passed to the RES is re-established on the remote host, and the task's status and resource usage are passed back to the client. Terminal I/O is transparent, so even applications such as vi that do complicated terminal manipulation run transparently on remote hosts. UNIX signals are supported across machines, so remote tasks get signals as if they were running locally. Job control also is done transparently. This level of transparency is maintained between heterogeneous hosts.

LSF batch system

LSF batch is a layered distributed load sharing batch system built on top of Platform LSF base. The services provided by LSF batch are extensions to the Platform LSF base services.

Application programmers can access batch services through the LSF batch Library (LSBLIB). The diagram below shows the components of LSF batch and their relationship:



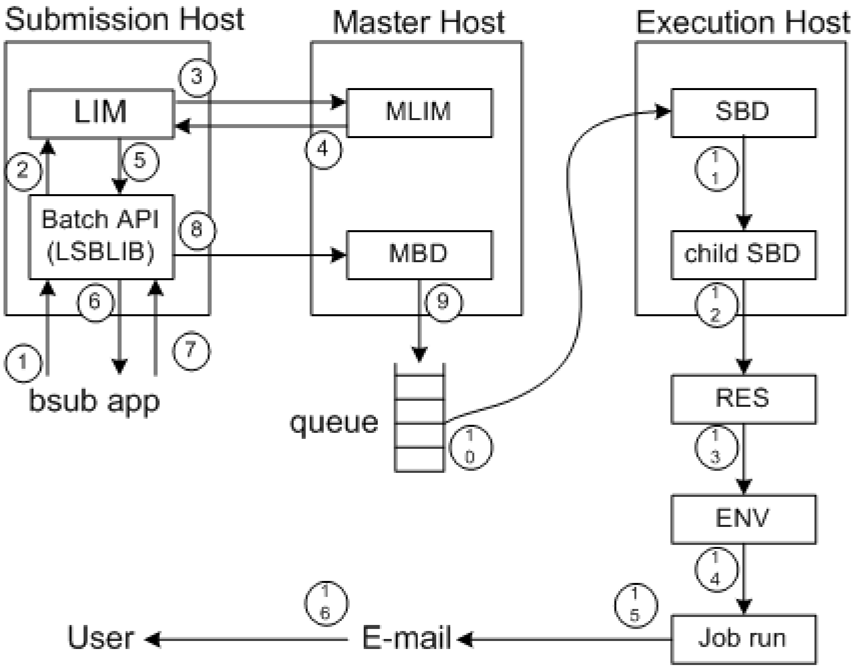
LSF batch accepts user jobs and holds them in queues until suitable hosts are available. LSF batch runs user jobs on LSF batch execution hosts, those hosts that a site deems suitable for running batch jobs.

LSBLIB consists of LSF API, the direct user interface to the rest of the LSF batch system. Platform LSF APIs provide easy access to the services of Platform LSF servers. The API routines hide the interaction details between the application and Platform LSF servers in a way that is platform independent.

LSF batch services are provided by two daemons, one `mbatchd` (master batch daemon) running in each Platform LSF cluster, and one `sbatchd` (slave batch daemon) running on each batch server host.

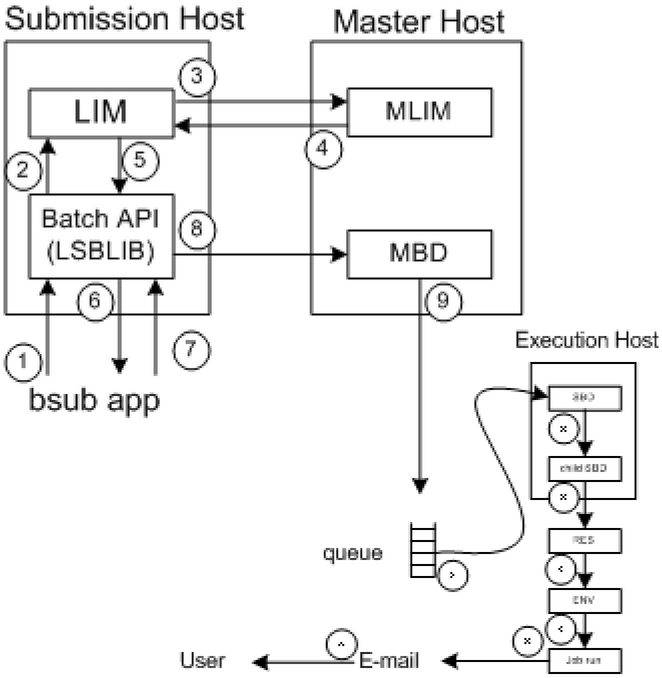
Application and Platform LSF batch interactions

LSF batch operation relies on the services provided by Platform LSF base. LSF batch contacts the master LIM to get load and resource information about every batch server host. The diagram below shows the typical operation of LSF batch:

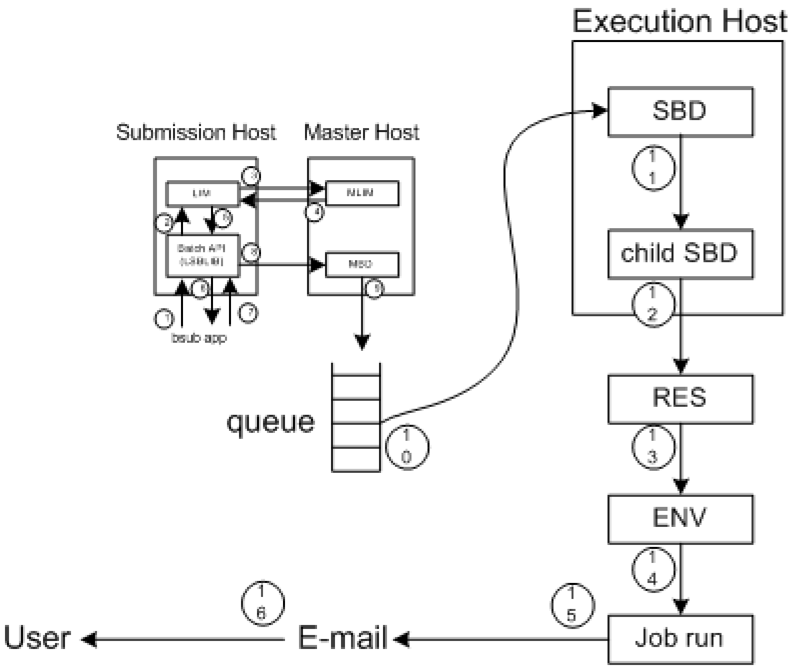


LSF batch executes jobs by sending user requests from the submission host to the master host. The master host puts the job in a queue and dispatches the job to an execution host. The job is run and the results are emailed to the user.

Unlike LSF base, the submission host does not directly interact with the execution host.



1. `bsub` or `lsb_submit()`submits a job to LSF for execution.
2. To access LSF base services, the submitted job proceeds through the Platform LSF batch library (LSBLIB) that contains LSF base library information.
3. The LIM communicates the job's information to the cluster's master LIM. Periodically, the LIM on individual machines gathers its 12 built-in load indices and forwards this information to the master LIM.
4. The master LIM determines the best host to run the job and sends this information back to the submission host's LIM.
5. Information about the chosen execution host is passed through the LSF batch library.
6. Information about the host to execute the job is passed back to `bsub` or `lsb_submit()`.
7. To enter the batch system, `bsub` or `lsb_submit()`sends the job to LSBLIB.
8. Using LSBLIB services, the job is sent to the `mbatchd` running on the cluster's master host.
9. The `mbatchd` puts the job in an appropriate queue and waits for the appropriate time to dispatch the job. User jobs are held in batch queues by `mbatchd`, which checks the load information on all candidate hosts periodically.



10. The `mbatchd` dispatches the job when an execution host with the necessary resources becomes available where it is received by the host's sbatchd. When more than one host is available, the best host is chosen.
11. Once a job is sent to an sbatchd, that sbatchd controls the execution of the job and reports the job's status to `mbatchd`. The sbatchd creates a child sbatchd to handle job execution.
12. The child sbatchd sends the job to the RES.
13. The RES creates the execution environment to run the job.
14. The job is run in the execution environment.
15. The results of the job are sent to the email system.
16. The email system sends the job's results to the user.

The `mbatchd` always runs on the host where the master LIM runs. The sbatchd on the master host automatically starts the `mbatchd`. If the master LIM moves to a different host, the current `mbatchd` will automatically resign and a new `mbatchd` will be automatically started on the new master host.

The log files store important system and job information so that a newly started `mbatchd` can restore the status of the previous `mbatchd`. The log files also provide historic information about jobs, queues, hosts, and LSF batch servers.

Platform LSF API services

Platform LSF services are natural extensions of operating system services. Platform LSF services glue heterogeneous operating systems into a single, integrated computing system.

Platform LSF APIs provide easy access to the services of Platform LSF servers.

Platform LSF APIs have been used to build numerous load sharing applications and utilities. Some examples of applications built on top of the Platform LSF APIs are `lsmake`, `lstcsh`, `lsrun`, and the LSF batch user interface.

Platform LSF base API services

The Platform LSF base API (LSLIB) allows application programmers to get services provided by LIM and RES. The services include:

- Configuration information service
- Dynamic load information service
- Placement advice service
- Task list information service
- Master Selection service
- Remote execution service
- Remote file operation service
- Administration service

Configuration information service

This set of function calls provide information about the Platform LSF cluster configuration, such as hosts belonging to the cluster, total amount of installed resources on each host (e.g., number of CPUs, amount of physical memory, and swap space), special resources associated with individual hosts, and types and models of individual hosts.

Such information is static and is collected by LIMs on individual hosts. By calling these routines, an application gets a global view of the distributed system. This information can be used for various purposes. For example, the Platform LSF command `lshosts` displays such information on the screen. LSF batch also uses such information to know how many CPUs are on each host.

Flexible options are available for an application to select the information that is of interest to it.

Dynamic load information service

This set of function calls provide comprehensive dynamic load information collected from individual hosts periodically. The load information is provided in the form of load indices detailing the load on various resources of each host, such as CPU, memory, I/O, disk space, and interactive activities. Since a site-installed External LIM (ELIM) can be optionally plugged into the LIM to collect additional information that is not already collected by the LIM, this set of services can be used to collect virtually any type of dynamic information about individual hosts.

Example applications that use such information include `lsload` and `lsmon`. This information is also valuable to an application making intelligent job scheduling decisions. For example, LSF batch uses such information to decide whether or not a job should be sent to a host for execution.

These service routines provide powerful mechanism for selecting the information that is of interest to the application.

Placement advice service

Platform LSF base API provides functions to select the best host among all the hosts. The selected host can then be used to run a job or to login. Platform LSF provides flexible syntax for an application to specify the resource requirements or criteria for host selection and sorting.

Many Platform LSF utilities use these functions for placement decisions, such as `lsrun`, `lsmake`, and `lslogin`. It is also possible for an application to get the detailed load information about the candidate hosts together with a preference order of the hosts.

A parallel application can ask for multiple hosts in one LSLIB call for the placement of a multi-component job.

The performance differences between different models of machines as well as the number of CPUs on each host are taken into consideration when placement advice is made, with the goal of selecting qualified hosts that will provide the best performance.

Task list manipulation service

Task lists are used to store default resource requirements for users. Platform LSF provides functions to manipulate the task lists and retrieve resource requirements for a task. This is important for applications that need to automatically pick up the resource requirements from user's task list. The Platform LSF command `lsrtasks` uses these functions to manipulate user's task list. Platform LSF utilities such as `lstcsh`, `lsrun`, and `bsub` automatically pick up the resource requirements of the submitted command line by calling these LSLIB functions.

Master selection service

If your application needs some kind of fault tolerance, you can make use of the master selection service provided by the LIM. For example, you can run one copy of your application on every host and only allow the copy on the master host to be the primary copy and others to be backup copies. LSLIB provides a function that tells you the name of the current master host.

LSF batch uses this service to achieve improved availability. As long as one host in the Platform LSF cluster is up, LSF batch service will continue.

Remote execution service

The remote execution service provides a transparent and efficient mechanism for running sequential as well as parallel jobs on remote hosts. The services are provided by the RES on the remote host in cooperation with the Network I/O Server (NIOS) on the local host. The NIOS is a per application stub process that handles the details of the terminal I/O and signals on the local side. NIOS is always automatically started by the LSLIB as needed.

RES runs as root and runs tasks on behalf of all users in the Platform LSF cluster. Proper authentication is handled by RES before running a user task.

Platform LSF utilities such as `lsrun`, `lsgrun`, `ch`, `lsmake`, and `lstcsh` use the remote execution service.

Remote file operation service

The remote file operation service allows load sharing applications to operate on files stored on remote machines. Such services extend the UNIX and Windows file operation services so that files that are not shared among hosts can also be accessed by distributed applications transparently.

LSLIB provides routines that are extensions to the UNIX and Windows file operations such as `open(2)`, `close(2)`, `read(2)`, `write(2)`, `fseek(3)`, `stat(2)`, etc.

The Platform LSF utility `lsrcp` is implemented with the remote file operation service functions.

Administration service

This set of function calls allow application programmers to write tools for administrating the Platform LSF servers. The operations include reconfiguring the Platform LSF clusters, shutting down a particular Platform LSF server on some host, restarting an Platform LSF server on some host, turning logging on or off, locking/unlocking a LIM on a host, etc.

The `lsadmin` utility uses the administration services.

LSF batch API services

The LSF batch API, LSBLIB, gives application programmers access to the job queueing processing services provided by the LSF batch servers. All LSF batch user interface utilities are built on top of LSBLIB. The services that are available through LSBLIB include:

- LSF batch system information service
- Job manipulation service
- Log file processing service
- LSF batch administration service

LSF batch system information service

This set of function calls allow applications to get information about LSF batch system configuration and status. These include host, queue, and user configurations and status.

The batch configuration information determines the resource sharing policies that dictate the behavior of the LSF batch scheduling.

The system status information reflects the current status of hosts, queues, and users of the LSF batch system.

Example utilities that use the LSF batch configuration information services are `bhosts`, `bqueues`, `busers`, and `bparams`.

Job manipulation service

The job manipulation service allows LSF batch application programmers to write utilities that operate on user jobs. The operations include job submission, signaling, status checking, checkpointing, migration, queue switching, and parameter modification.

Log file processing service

Log file events can be used to produce historical information about the LSF batch system and user jobs. Such information can be used to produce accounting or statistic reports.

Examples of utilities that use log file processing are `bacct` and `bhist`.

LSF batch administration service

This set of function calls are useful for writing LSF batch administration tools.

The LSF batch command `badmin` is implemented with these library calls.

Platform LSF programs

Platform LSF programming is like any other system programming. You are assumed to have UNIX and/or Windows operating system and C programming knowledge to understand the concepts involved in this section.

lsf.conf file

This guide frequently refers to the file, `lsf.conf`, for the definition of some parameters. `lsf.conf` is a generic reference file containing definitions of directories and parameters. It is by default installed in `/etc`. If it is not installed in `/etc`, all users of Platform LSF must set the environment variable LSF_ENVDIR to point to the directory in which `lsf.conf` is installed. See the Platform LSF Reference for more details about the `lsf.conf` file.

Platform LSF header files

All Platform LSF header files are installed in the directory `LSF_INCLUDEDIR/lsf`, where `LSF_INCLUDEDIR` is defined in the file `lsf.conf`. You should include `LSF_INCLUDEDIR` in the include file search path, such as that specified by the ‘`-Idir`’ option of some compilers or pre-processors.

There is one header file for LSLIB, the Platform LSF base API, and one header file for LSBLIB, the LSF batch API.

lsf.h

An Platform LSF application must include `<lsf/lsf.h>` before any of the Platform LSF base API services are called. `lsf.h` contains definitions of constants, data structures, error codes, LSLIB function prototypes, macros, etc., that are used by all Platform LSF applications.

lsbatch.h

An LSF batch application must include `<lsf/lsbatch.h>` before any of the LSF batch API services are called. `lsbatch.h` contains definitions of constants, data structures, error codes, LSBLIB function prototypes, macros, etc., that are used by all LSF batch applications.

Tip:

There is no need to explicitly include `<lsf/lsf.h>` in an LSF batch application because `lsbatch.h` includes `<lsf/lsf.h>`.

Link applications with Platform LSF APIs

For all UNIX platforms, Platform LSF API functions are contained in two libraries, `liblsf.a` (LSLIB) and `libbat.a` (LSBLIB). For Windows, the file names of these libraries are: `liblsf.lib` (LSLIB) and `libbat.lib` (LSBLIB). These files are installed in `LSF_LIBDIR`, where `LSF_LIBDIR` is defined in the file `lsf.conf`.

Note:

LSBLIB is not independent. It must always be linked together with LSLIB because LSBLIB services are built on top of LSLIB services.

Platform LSF uses BSD sockets for communication across a network. On systems that have both System V and BSD programming interfaces, LSLIB and LSBLIB typically use the BSD programming interface. On System V-based versions of UNIX such as Solaris, it is necessary to link applications using LSLIB or LSBLIB with the BSD compatibility library. On Windows, a number of libraries need to be linked together with LSF API. Details of these additional linkage specifications (libraries and link flags) are shown in the table below.

Platform	**Additional Linkage Specifications**
ULTRIX 4	None
Digital UNIX	`-lmach -lmld`
HP-UX	`-lBSD`
AIX	`-lbsd`
IRIX 5	`-lsun -lc_s`
IRIX 6	None
SunOS 4	None
Solaris 2	`-lnsl -lelf -lsocket -lrpcsvc -lgen -ldl`
Solaris 7 32-bit	`-lnsl -lefl -lsocket -lrpcsvc -lgen -ldl -DSVR4 -lresolv -lm`
Solaris 7 64-bit	`-lnsl -lefl -lsocket -lrpcsvc -lgen -ldl -Xarch=v9 -lresolv -lm`
NEC	`-lnsl -lelf -lsocket -lrpcsvc -lgen`
Sony NEWSs	`-lc -lnsl -lelf -lsocket -lrpcsvc -lgen -lucb`
ConvexOS	None
Cray Unicos	None
Linux	libnsl.a
Windows 2000	`-MT -DWIN32 libcmt.lib oldnames.lib kernel32.lib advapi32.lib user32.lib wsock32.lib mpr.lib netapi32.lib userenv.lib` oleaut32.lib uuid.lib activeds.lib adsiid.lib ole32.lib liblsf.lib libbat.lib
Windows XP	`-MT -DWIN32 libcmt.lib oldnames.lib kernel32.lib advapi32.lib user32.lib wsock32.lib mpr.lib netapi32.lib userenv.lib` oleaut32.lib uuid.lib activeds.lib adsiid.lib ole32.lib liblsf.lib libbat.lib

Note:

On Windows, you need to add paths specified by `LSF_LIBDIR` and `LSF_INCLUDEDIR` in `lsf.conf` to the environment variables LIB and INCLUDE.

Recall that the GNU C compiler on Solaris only supports 32 bit application development (not 64 bit). Link your 32 bit applications on Solaris with the 32 bit LSF sparc-sol7-32 distribution file.

The `$LSF_MISC/examples` directory contains a makefile for making all the example programs in that directory. You can modify this file and the example programs for your own use.

All LSLIB function call names start with `ls_`.

All LSBLIB function call names start with `lsb_`.

Compile LSF API programs

Compile an LSF API program without using the makefile.

1. Include the LSF API libraries and the link flags for the appropriate architecture on the command line.

   This establishes the compilation environment.

   For example, to compile an LSF API program on a Solaris 2.x 32 bit machine, you will have a compilation statement similar to the following:

   ```
   % cc -o simbhosts simbhosts.c -I$LSF_ENVDIR/../include
   $LSF_LIBDIR/libbat.a $LSF_LIBDIR/liblsf.a -lnsl -lelf
   -lsocket -lrpcsvc -lgen -ldl -lresolv -lm
   ```

   - The flag `-I$LSF_ENVDIR/../include` specifies the location of the LSF include directory.
   - `$LSF_LIBDIR/libbat.a` and `$LSF_LIBDIR/liblsf.a` are the locations of the LSLIB and LSBLIB.
   - We include the following extra compilation flags as given from the above chart:

     ```
     -lnsl -lelf -lsocket -lrpcsvc -lgen -ldl -lresolv -
     lm
     ```
   - The resulting executable of the program `simbhosts.c` is called `simbhosts`.

Compile an LSF API program on a 64 bit Solaris 2.x

1. Add the xarch setting as follows:

   ```
   % cc -xarch=v9 -o simbhosts simbhosts.c -I$LSF_ENVDIR/../include
   $LSF_LIBDIR/libbat.a $LSF_LIBDIR/liblsf.a -lnsl -lelf -lsocket -lrpcsvc -
   lgen -ldl -lresolv -lm
   ```

Compile on Linux

1. Include the `libnsl.a` library.

   This library is located in `/usr/lib/`.

   For example, when compiling a program on redhat6.2-intel, use the following:

   ```
   gcc program.c -I$LSF_ENVDIR/../include $LSF_LIBDIR/libbat.a
   $LSF_LIBDIR/liblsf.a $LSF_LIBDIR/libnsl.a -lm -lnsl -ldl
   ```

   where `program.c` is the name of the program you want to compile.

Compile on Solaris x86-64-sol10

1. Use the following library and link flags:

```
/opt/SUNWspro/bin/cc obj.c -R/usr/dt/lib:/usr/openwin/lib -DSVR4 -DSOLARIS
-DSOLARIS64 -xs -xarch=amd64 -D_TS_ERRNO -Dx86_64 -DSOLARIS2_5 -DSOLARIS2_7
-DI18N_COMPILE -DSOLARIS2_8 -DSOLARIS2_10 -DSTD_SHARED_OBJ -lbat -llsf -lnsl
-lelf -lsocket -lrpcsvc -lgen -ldl -lresolv -o obj_name
```

where `obj.c` is the name of the program you want to compile and *obj_name* is the name of the binary you can run after compiling the program.

Set up Visual Studio

You can use Visual Studio 2005 or 2008 to build the application with LSF APIs.

1. Create a Win32 Console project.
2. Add a test program as the source file `test.c` .

```
Test Program:
---------------------------------------------------------------
#include <stdio.h>
#include <lsf/lsf.h>
void main()
{
    char *clustername;
    clustername = ls_getclustername();
    if (clustername == NULL) {
        ls_perror("ls_getclustername");
        exit(-1);
    }
    printf("My cluster name is: <%s>\n", clustername);
    exit(0);
}
```

3. Add the LSF 7.0 `include` and `lib` directories as additional include and library directories.
4. Add the following `lib` files as additional dependencies:
    - `oldnames.lib`
    - `mpr.lib`
    - `netapi32.lib`
    - `userenv.lib`
    - `activeds.lib`
    - `adsiid.lib`
    - `liblsf.lib`
    - `libbat.lib`
    - `WSOCK32.lib`
    - `WS2_32.lib`
    - `MSWSOCK.lib`
5. Include any special build options, as required.

    For example, in Visual C++ 2005, the size of the time_t data type was changed from 32 bits to 64 bits. However, the LSF package is built with VC60 (in which size of time_t data type is 32 bits). To solve, choose one of the two following solutions:

    - In Visual Studio, change the C/C++ command line additional options to include -D "_USE_32BIT_TIME_T".
    - Add one line to the beginning of `stdafx.h`.

```
#define _USE_32BIT_TIME_T
```

To change the build environment from 32 to 64 bits, add the build option -D"WIN32".

Error handling

Platform LSF API uses error numbers to indicate an error. There are two global variables that are accessible from the application. These variables are used in exactly the same way UNIX system call error number variable `errno` is used. The error number should only be tested when an LSLIB or LSBLIB call fails.

lserrno

An Platform LSF program should test whether an LSLIB call is successful or not by checking the return value of the call instead of `lserrno`.

When any LSLIB function call fails, it sets the global variable `lserrno` to indicate the cause of the error. The programmer can either call `ls_perror()` to print the error message explicitly to the `stderr`, or call `ls_sysmsg()` to get the error message string corresponding to the current value of `lserrno`.

Possible values of `lserrno` are defined in `lsf.h`.

lsberrno

This variable is very similar to `lserrno` except that it is set by LSBLIB whenever an LSBLIB call fails. Programmers can either call `lsb_perror()` to find out why an LSBLIB call failed or use `lsb_sysmsg()` to get the error message corresponding to the current value of `lsberrno`.

Possible values of `lsberrno` are defined in `lsbatch.h`.

Tip:

`lserrno` should be checked only if an LSLIB call fails. If an LSBLIB call fails, then `lsberrno` should be checked .

Example applications

Example application using LSLIB

```
#include <stdio.h>
#include <lsf/lsf.h>
void main()
{
    char *clustername;
    clustername = ls_getclustername();
    if (clustername == NULL) {
        ls_perror("ls_getclustername");
        exit(-1);
    }
    printf("My cluster name is: <%s>\n", clustername);
    exit(0);
}
```

This simple example gets the name of the Platform LSF cluster and prints it on the screen. The LSLIB function call `ls_getclustername()` returns the name of the local cluster. If this call fails, it returns a `NULL` pointer. `ls_perror()` prints the error message corresponding to the most recently failed LSLIB function call.

The above program would produce output similar to the following:

```
% a.out
My cluster name is: <test_cluster>
```

Example application using LSBLIB

```
#include <stdio.h>
#include<lsf/lsbatch.h>
int main()
{
    struct parameterInfo *parameters;
    if (lsb_init(NULL) < 0) {
        lsb_perror("lsb_init");
        exit(-1);
    }
    parameters = lsb_parameterinfo(NULL, NULL, 0);
    if (parameters == NULL) {
        lsb_perror("lsb_parameterinfo");
        exit(-1);
    }
    /* Got parameters from mbatchd successfully. Now print out     the fields */
    printf("Job acceptance interval: every %d dispatch turns\n",parameters->jobAcceptInterval);
    /* Code that prints other parameters goes here */
        /* ... */
    exit(0);
}
```

This example gets the LSF batch parameters and prints them on the screen. The function `lsb_init()` must be called before any other LSBLIB function is called.

The data structure parameterInfo is defined in `lsbatch.h`.

Authentication

Platform LSF programming is distributed programming. Since Platform LSF services are provided network-wide, it is important for Platform LSF to deliver the service without compromising the system security.

Platform LSF supports several user authentication protocols. Support for these protocols are described in Administering Platform LSF. Your Platform LSF administrator can configure the Platform LSF cluster to use any of the supported protocols.

Only those Platform LSF API function calls that operate on user jobs, user data, or Platform LSF servers require authentication. Function calls that return information about the system do not need to be authenticated.

The most commonly used authentication protocol, the privileged port protocol, requires that load sharing applications be installed as setuid programs. This means that your application has to be owned by root with the setuid bit set. .

If you need to frequently change and re-link your applications with Platform LSF API, you can consider using the ident protocol which does not require applications to be setuid programs.

CHAPTER

2

Programming with LSLIB

Configuration information

One of the services that LSF provides to applications is cluster configuration information. This section describes how to get this service with a C program using LSLIB.

General cluster configuration information

In the previous chapter, a very simple application was introduced that prints the name of the LSF cluster. This section extends that example by printing the current master host name and the defined resource names in the cluster. It uses the following additional LSLIB function calls:

```
struct lsInfo *ls_info(void)
char *ls_getclustername(void)
char *ls_getmastername(void)
```

All of these functions return NULL on failure and set `lserrno` to indicate the error.

lsinfo structure

The function `ls_info()` returns a pointer to the lsinfo data structure

(defined in `<lsf/lsf.h>`):

```
struct lsInfo {
   int   nRes;                  Number of resources in the system
   struct resItem *resTable;     A resItem for each resource in the system
   int   nTypes;                Number of host types
   char  hostTypes[MAXTYPES][MAXLSFNAMELEN];   Host types
   int   nModels;               Number of host models
   char  hostModels[MAXMODELS][AXLSFNAMELEN];  Host models
   char  hostArchs[MAXMODELS][MAXLSFNAMELEN];  Architecture name
   int   modelRefs[MAXMODELS]; Number of hosts of this architecture
   float cpuFactor[MAXMODELS];  CPU factors of each host model
   int   numIndx;               Total number of load indices in resItem
   int   numUsrIndix;           Number of user-defined load indices
};
```

resItem structure

Within `struct lsinfo`, the resItem data structure describes the valid resources defined in the LSF cluster:

```
struct resItem {
   char name[MAXLSFNAMELEN];  The name of the resource
   char des[MAXRESDESLEN];    The description of the resorce
   enum valueType valueType; Type of value: BOOLEAN,
NUMERIC,                                STRING, EXTERNAL
   enum orderType orderType;  Order: INCR, DECR, NA
   int  flags;             Resource attribute flags
#define RESF_BUILTIN 0x01   Built-in vs configured resource
#define RESF_DYNAMIC 0x02   Dynamic vs static value
#define RESF_GLOBAL 0x04    Resource defined in all clusters
#define RESF_SHARED 0x08    Shared resource for some hosts
#define RESF_LIC 0x10       License static value
#define RESF_EXTERNAL 0x20  External resource defined
#define RESF_RELEASE 0x40   Resource can be released when job is
                            suspended
   int interval;            The update interval for a load index, in seconds
};
```

The constants `MAXTYPES`, `MAXMODELS`, and `MAXLSFNAMELEN` are defined in `<lsf/lsf.h>`. `MAXLSFNAMELEN` is the maximum length of a name in LSF.

A host type in LSF refers to a class of hosts that are considered to be compatible from an application point of view. This is entirely configurable, although normally hosts with the same architecture (binary compatible hosts) should be configured to have the same host type.

A host model in LSF refers to a class of hosts with the same CPU performance. The CPU factor of a host model should be configured to reflect the CPU speed of the model relative to other host models in the LSF cluster.

`ls_getmastername()` returns a string containing the name of the current master host.

`ls_getclustername()` returns a string containing the name of the local load sharing cluster defined in the configuration files.

The returned data structure of every LSLIB function is dynamically allocated inside LSLIB. This storage space is automatically freed by LSLIB and re-allocated next time the same LSLIB function is called. An application should never attempt to free the storage returned by LSLIB. If you need to keep this information across calls, make your own copy of the data structure. This applies to all LSLIB function calls.

Example

The following program displays LSF cluster information using the above LSLIB function calls.

```
#include <stdio.h>
#include <lsf/lsf.h>

main()
{
    struct lsInfo *lsInfo;
    char *cluster, *master;
    int i;
/* get the name of the local load sharing cluster */
    cluster = ls_getclustername();
    if (cluster == NULL) {
        ls_perror("ls_getclustername");
        exit(-1);
    }
    printf("My cluster name is <%s>\n", cluster);
/* get the name of the current master host */
    master = ls_getmastername();
    if (master == NULL) {
        ls_perror("ls_getmastername");
        exit(-1);
    }
    printf("Master host is <%s>\n", master);
/* get the load sharing configuration information */
    lsInfo = ls_info();
    if (lsInfo == NULL) {
        ls_perror("ls_info");
        exit(-1);
    }
    printf("\n%-15.15s %s\n", "RESOURCE_NAME", "DESCRIPTION");
    for (i=0; i<lsInfo->nRes; i++)
        printf("%-15.15s %s\n",
            lsInfo->resTable[i].name, lsInfo->resTable[i].des);

    exit(0);
}
```

The above program will produce output similar to the following:

```
% a.out
My cluster name is <test_cluster>
Master host is <hostA>

RESOURCE_NAME      DESCRIPTION
r15s               15-second CPU run queue length
r1m                1-minute CPU run queue length (alias: cpu)
```

```
r15m               15-minute CPU run queue length
ut                 1-minute CPU utilization (0.0 to 1.0)
pg                 Paging rate (pages/second)
io                 Disk IO rate (Kbytes/second)
ls                 Number of login sessions (alias: login)
it                 Idle time (minutes) (alias: idle)
tmp                Disk space in /tmp (Mbytes)
swp                Available swap space (Mbytes) (alias: swap)
mem                Available memory (Mbytes)
ncpus              Number of CPUs
ndisks             Number of local disks
maxmem             Maximum memory (Mbytes)
maxswp             Maximum swap space (Mbytes)
maxtmp             Maximum /tmp space (Mbytes)
cpuf               CPU factor
rexpri             Remote execution priority
server             LSF server host
LSF_Base           Base product
lsf_base           Base product
LSF_Manager        Standard product
lsf_manager        Standard product
LSF_JobSchedule    JobScheduler product
lsf_js             JobScheduler product
LSF_Make           Make product
lsf_make           Make product
LSF_Parallel       Parallel product
lsf_parallel       Parallel product
LSF_Analyzer       Analyzer product
lsf_analyzer       Analyzer product
mips               MIPS architecture
dec                DECStation system
sparc              SUN SPARC
bsd                BSD unix
sysv               System V UNIX
hpux               HP-UX UNIX
aix                AIX UNIX
irix               IRIX UNIX
ultrix             Ultrix UNIX
solaris            SUN SOLARIS
sun41              SunOS4.1
convex             ConvexOS
osf1               OSF/1
fs                 File server
cs                 Compute server
frame              Hosts with FrameMaker license
bigmem             Hosts with very big memory
diskless           Diskless hosts
alpha              DEC alpha
linux              LINUX UNIX
type               Host type
model              Host model
status             Host status
hname              Host name
```

Host configuration information

Host configuration information describes the static attributes of individual hosts in the LSF cluster. Examples of such attributes are host type, host model, number of CPUs, total physical memory, and the special resources associated with the host. These attributes are either read from the LSF configuration file, or determined by the host's LIM on start up.

ls_gethostinfo()

Host configuration information can be obtained by calling `ls_gethostinfo()`:

```
struct hostInfo *ls_gethostinfo(resreq, numhosts, hostlist,
                                 listsize, options)
```

`ls_gethostinfo()` has these parameters:

```
char *resreq;       Resource requirements that a host must satisfy
int *numhosts;      The number of hosts
char **hostlist;    An array of candidate hosts
int listsize;       Number of candidate hosts
int options;        Options, currently only DFT_FROMTYPE
```

On success, `ls_gethostinfo()` returns an array containing a hostInfo structure for each host. On failure, it returns `NULL` and sets `lserrno` to indicate the error.

hostInfo structure

The `hostInfo` structure is defined in `lsf.h` as

```
struct hostInfo {
    char  hostName[MAXHOSTNAMELEN];
    char  *hostType;
    char  *hostModel;
    float cpuFactor;
    int   maxCpus;
    int   maxMem;
    int   maxSwap;
    int   maxTmp;
    int   nDisks;
    int   nRes;
    char  **resources;
    int   nDRes;
    char  **DResources;
    char  *windows;
    int   numIndx;
    float *busyThreshold;
    char  isServer;
    char  licensed;
    int   rexPriority;
    int   licFeaturesNeeded;
#define LSF_BASE_LIC  0
#define LSF_BATCH_LIC_OBSOLETE 1
#define LSF_JS_SCHEDULER_LIC  2
#define LSF_JS_LIC  3
#define LSF_CLIENT_LIC  4
#define LSF_MC_LIC  5
#define LSF_ANALYZER_SERVER_LIC  6
#define LSF_MAKE_LIC  7
#define LSF_PARALLEL_LIC         8
#define LSF_FLOAT_CLIENT_LIC     9
#define LSF_FTA_LIC              10
#define LSF_AFTER_HOURS_LIC 11
#define LSF_RESOURCE_PREEMPT_LIC 12
#define LSF_BACCT_LIC            13
#define LSF_SCHED_FAIRSHARE_LIC  14
#define LSF_SCHED_RESERVE_LIC    15
#define LSF_SCHED_PREEMPTION_LIC 16
#define LSF_SCHED_PARALLEL_LIC   17
#define LSF_SCHED_ADVRSV_LIC     18
#define LSF_API_CLIENT_LIC       19
#define CLUSTERWARE_MANAGER_LIC 20
#define LSF_MANAGER_LIC  21
#define LSF_PCC_HPC_LIC          22 /*"platform_hpc" feature*/
#define sCLUSTERWARE_LIC  23 /*"s-Clusterware" OEM for S&C */
#define OTTAWA_MANAGER_LIC       24
#define SYMPHONY_MANAGER_ONLINE_LIC 25
#define SYMPHONY_MANAGER_BATCH_LIC  26
#define SYMPHONY_SCHED_JOB_PRIORITY_LIC 27
#define LSF_DUALCORE_X86_LIC     28
#define LSF_TSCHED_LIC           29
#define LSF_NUM_LIC_TYPE         30
#define LSF_NO_NEED_LIC          32
    int licClass;                /*license class needed */
    int cores;                   /* number of cores per physical CPU */
#ifndef INET6_ADDRSTRLEN
# define INET6_ADDRSTRLEN 46
#endif
    char hostAddr[INET6_ADDRSTRLEN]; /* IP address of this host */
    int pprocs;                    /* 82185 - Num physical processors. */
    /* cores_per_proc and cores are both needed for backwards compatibility.
    * cores is used for licencing enforcement and cores_per_proc is needed
    * for ncpus computation. */
    int cores_per_proc;            /* 82185 - Num cores per processor. */
   int threads_per_core;         /* 82185 - Num threads per core. */
};
```

Tip:

On Solaris, when referencing `MAXHOSTNAMELEN`, `netdb.h` must be included before `lsf.h` or `lsbatch.h`.

`NULL` and `0` were supplied for the hostlist and listsize parameters of the `ls_gethostinfo()` call. This causes all LSF hosts meeting resreq to be returned. If a host list parameter is supplied with this call, the selection of hosts will be limited to those belonging to the list.

If `resreq` is `NULL`, then the default resource requirements will be used.

The values of maxMem and maxCpus (along with maxSwap, maxTmp, and nDisks) are determined when LIM starts on a host. If the host is unavailable, the master LIM supplies a negative value.

Example

The following example shows how to use `ls_gethostinfo()` in a C program. It displays the name, host type, total memory, number of CPUs and special resources for each host that has more than 50MB of total memory.

```
#include <netdb.h>    /* Required for Solaris to reference
                      MAXHOSTNAMELEN */
#include <lsf/lsf.h>
#include <stdio.h>

main()
{
    struct hostInfo *hostinfo;
    char   *resreq;
    int    numhosts = 0;
    int    options = 0;
    int    i, j;
/* only hosts with maximum memory larger than 50 Mbytes are of interest */
   resreq="maxmem>50";
/* get information on interested hosts */
   hostinfo = ls_gethostinfo(resreq, &numhosts, NULL, 0, options);
    if (hostinfo == NULL) {
        ls_perror("ls_gethostinfo");
        exit(-1);
    }
/* print out the host names, host types, maximum memory, number of CPUs and
number of resources */
    printf("There are %d hosts with more than 50MB total memory          \n
\n", numhosts);
    printf("%-11.11s %8.8s %6.6s %6.6s %9.9s\n",
           "HOST_NAME", "type", "maxMem", "ncpus", "RESOURCES");

    for (i = 0; i < numhosts; i++) {
        printf("%-11.11s %8.8s", hostinfo[i].hostName,
                hostinfo[i].hostType);

        if (hostinfo[i].maxMem > 0)
            printf("%6dM ", hostinfo[i].maxMem);
       else                         /* maxMem info not available for this host*/
            printf("%6.6s ", "-");

        if (hostinfo[i].maxCpus > 0)
```

```
            printf("%6d ", hostinfo[i].maxCpus);
        else              /* ncpus is not known for this host*/
            printf("%6.6s", "-");

        for (j = 0; j < hostinfo[i].nRes; j++)
            printf(" %s", hostinfo[i].resources[j]);

        printf("\n");
    }
    exit(0);
}
```

In the above example, resreq defines the resource requirements used to select the hosts. The variables you can use for resource requirements must be the resource names returned from `ls_info()`. You can run the `lsinfo` command to obtain a list of valid resource names in your LSF cluster.

The above example program produces output similar to the following:

```
% a.out
There are 4 hosts with more than 50MB total memory

HOST_NAME   type     maxMem ncpus RESOURCES
hostA      HPPA10    128M    1    hppa hpux cs
hostB      ALPHA      58M    2    alpha cs
hostD      ALPHA      72M    4    alpha fddi
hostC      SUNSOL     54M    1    solaris fddi
```

To get specific host information use:

- `char *ls_gethosttype(`*hostname*`)`
- `Returns the type of a specific host`
- `char *ls_gethostmodel(`*hostname*`)`
- `Returns the model of a specific host`
- `float *ls_gethostfactor(`*hostname*`)`
- `Returns the CPU factor of the specified host`

Manage hosts

Using LSF base APIs you can manage hosts in your cluster by:

- Removing hosts from a cluster
- Adding hosts to a cluster
- Locking a host in a cluster
- Unlocking a host in a cluster

To manage the hosts in your cluster you need to be root or the LSF administrator as defined in the file:

`LSF_CONFDIR/lsf.cluster.<clustername>`

By managing your hosts you can control the placement of jobs and manage your resources more effectively.

Remove hosts from a cluster

Before you remove a host from a cluster, you need to shut down the host’s LIM.

1. Shut down the host’s LIM, using `ls_limcontrol()`:

   int ls_limcontrol (char *hostname, int opCode)

   `ls_limcontrol()`has the following parameters:

- char *hostname: the host’s name
- int opCode: operation code

where `opCode` describes the `ls_limcontrol()`operation. To shut down a host’s LIM, choose the following operation code:

LIM_CMD_SHUTDOWN

Example

The following code example demonstrates how to shut down a host’s LIM using `ls_limcontrol()`:

```
/******************************************************
* LSLIB -- Examples
*
* ls_limcontrol()
* Shuts down or reboots a host’s LIM.
******************************************************/
#include <lsf/lsf.h>
#include <io.h>
#include <stdlib.h>
#include <stdio.h>
int main(int argc, char ** argv)
{
  int result; /* returned value from ls_limcontrol*/
  int opCode; /*option*/
  char* host; /*host*/
   /* Checking for the correct format */
  if (argc !=2)
{
      fprintf(stderr, "usage: sudo %s <host>\n", argv[0]);
      exit(-1);
}
host = argv[1];
/* To shut down a host, assign LIM_CMD_SHUTDOWN to the
opCode */
opCode = LIM_CMD_SHUTDOWN;
printf("Shutting down LIM on host <%s>\n", host);
result =ls_limcontrol(host, opCode);
/* If there is an Error in execution, the program exits */
if (result == -1)
{
    ls_perror("ls_limcontrol");
    exit(-1);
}
/* Otherwise, indicate successful program execution */
else
{
    printf("host <%s> shutdown successful.\n", host);
    exit (0);
}
```

To use the above example, at the command line type:

```
sudo ./a.out hostname
```

where `hostname` is the name of the host you want to move to another cluster.

Add hosts to a cluster

When you return a removed host to a cluster, you need to reboot the host’s LIM. When you reboot the LIM, the configuration files are read again and the previous LIM status of the host is lost.

1. To reboot a host’s LIM, use `ls_limcontrol()`:

   `int ls_limcontrol (char *hostname, int opCode)`
2. Choose the following operation code (`opCode`):

   `LIM_CMD_REBOOT`

Example

The following code example demonstrates how to reboot a host’s LIM using `ls_limcontrol()`:

```
/******************************************************
* LSLIB -- Examples
*
* ls_limcontrol()
* Shuts down or reboots a host’s LIM.
******************************************************/
#include <lsf/lsf.h>
#include <io.h>
#include <stdlib.h>
#include <stdio.h>
int main(int argc, char ** argv)
{
  int result; /* returned value from ls_limcontrol*/
  int opCode; /*option*/
  char* host; /*host*/
   /* Checking for the correct format */
  if (argc !=2)
{
      fprintf(stderr, "usage: sudo %s <host>\n", argv[0]);
      exit(-1);
}
host = argv[1];
/* To reboot a host, assign LIM_CMD_REBOOT to the opCode */
opCode = LIM_CMD_REBOOT;
printf("Restarting LIMon host <%s>\n", host);
result =ls_limcontrol(host, opCode);
/* If there is an Error in execution, the program exits */
if (result == -1)
{
    ls_perror("ls_limcontrol");
    exit(-1);
}
/* Otherwise, indicate successful program execution */
else
{
    printf("host <%s> has been rebooted. \n", host);
}
/*Reboot is successful and the program exits */
    exit (0);
}
```

To use the above example, at the command line type:

```
sudo ./a.out hostname
```

where `hostname` is the name of the host you want to return to a cluster.

Lock a host in a cluster

Locking a host prevents a host from being selected by the master LIM for task or job placement.

Locking a host is useful for managing your resources:

- You can isolate machines in your cluster and apply their resources to particular work.
- If machine owners want private control over their machines, you can allow this indefinitely or for a period of time that you choose.
- Hosts can be unlocked automatically or unlocked manually.

1. To lock a host, use `ls_lockhost()`:

   `int ls_lockhost(time_t duration)`

   `ls_lockhost()`has the following parameter:

   - time_t duration: The number of seconds the host is locked

   a) To lock a host indefinitely, assign 0 seconds to `duration`.

   b) To automatically unlock a host, assign a value greater than 0 to `duration` and the host will automatically unlock when time has expired.

Note:

If you try to lock a host that is already locked, `ls_lockhost()`sets `lserrno` to `LSE_LIM_ALOCKED`.

Example

The following code example demonstrates how to use `ls_lockhost()` to lock a host:

```
/*****************************************************
* LSLIB -- Examples
*
* ls_lockhost()
* Locks the local host for a specified time.
*****************************************************/
#include <lsf/lsf.h>
#include <time.h>
int main(int argc, char ** argv)
{
/* Declaring variables*/
   u_long duration;
/* Checking for the correct format */
   if (argc !=2)
{
     fprintf(stderr, "usage: sudo %s <duration>\n", argv
[0]);
     exit(-1);
}
/* assigning the duration of the lockage*/
   duration = atoi(argv[1]);
/* If an error occurs, exit with an error msg*/
   if (ls_lockhost(duration) !=0)
{
     ls_perror("ls_lockhost");
     exit(-1);
}
/* If ls_lockhost() is successful, then check to see if
duration is > 0. Indicate how long the host is locked if
duration is >0 */
   if (duration > 0)
{
     printf("Host is locked for %i seconds \n",
(int)              duration);
}
   else /* Indicate indefinite lock on host */
{
    printf("Host is locked\n");
}
/* successful exit */
exit(0);
}
```

Unlock a host in a cluster

Hosts that have been indefinitely locked by assigning the value 0 to the duration parameter of `ls_lockhost()` can only be manually unlocked.

1. To manually unlock a host, use `ls_unlockhost()`:

   int ls_unlockhost(void)

Note:

By unlocking a host, the master LIM can choose the host for task or job placement.

Example

The following code example demonstrates how to use `ls_unlockhost()` to manually unlock a host:

```
/******************************************************
* LSLIB -- Examples
*
* ls_unlockhost()
* Unlocks an indefinitely locked local host.
******************************************************/
#include <lsf/lsf.h>
#include <stdlib.h>
#include <stdio.h>
int main(int argc, char ** argv)
{
/* Checking for the correct format*/
   if (argc !=1)
{
     fprintf(stderr, "usage: sudo %s\n", argv[0]);
     exit(-1);
}
/* Call ls_unlockhost(). If an error occurs, print an error
msg and exit.*/
   if (ls_unlockhost() <0)
{
       ls_perror("ls_lockhost");
       exit(-1);
}
/* Indicate a successful ls_unlockhost() call and exit.*/
       printf("Host is unlocked\n");
       exit(0);
}
```

Default resource requirements

Some LSLIB functions require a resource requirement parameter. This parameter is passed to the master LIM for host selection. It is important to understand how LSF handles default resource requirements. See Administering Platform LSF for further information about resource requirements.

It is desirable for LSF to automatically assume default values for some key requirements if they are not specified by the user.

The default resource requirements depend on the specific application context. For example, the `lsload` command assumes ‘type==any order[r15s:pg]’ as the default resource requirements, while `lsrun` assumes ‘type==local order [r15s:pg]’ as the default resource requirements. This is because the user usually expects `lsload` to show the load on all hosts. With `lsrun`, a task using run on the same host type as the local host, causes the task to be run on the correct host type.

LSLIB provides the flexibility for the application programmer to set the default behavior.

LSF default resource requirements contain two parts, a type requirement and an order requirement. A type requirement ensures that the correct type of host is selected. Use an order requirement to order the selected hosts according to some reasonable criteria.

LSF appends a type requirement to the resource requirement string supplied by an application in the following situations:

- resreq is NULL or an empty string.
- resreq does not contain a boolean reasource, for example, ‘hppa’, and does not contain a *type* or model resource, for example, ‘type==solaris’, ‘model==HP715’.

The default type requirement can be either ‘type==any’ or ‘type==$fromtype’ depending on whether or not the flag DFT_FROMTYPE is set in the options parameter of the function call. DFT_FROMTYPE is defined in `lsf.h`.

If DFT_FROMTYPE is set in the options parameter, the default *type requirement* is ‘type==$fromtype’. If DFT_FROMTYPE is not set, then the default *type requirement* is ‘type==any’.

The value of fromtype depends on the function call. If the function has a fromhost parameter, then fromtype is the host type of the fromhost. fromhost is the host that submits the task. Otherwise, fromtype is local.

LSF also appends an *order requirement*, order[r15s:pg], to the resource requirement string if an *order requirement* is not already specified.

The table below lists some examples of how LSF appends the default resource requirements.

User’s Resource Requirement	**Resource Requirement After Appending the Default**	
	DFT_FROMTYPE set	**DFT_FROMTYPE not set**
NULL	type==$fromtype order[r15s:pg]	type==any order[r15s:pg]
hpux	hpux order[r15s:pg]	hpux order[r15s:pg]
order[r1m]	type==$fromtype order[r1m]	type==any order[r1m]
model==hp735	model==hp735 order[r15s:pg]	model==hp735 order[r15s:pg]
sparc order[ls]	sparc order[ls]	sparc order[ls]

User’s Resource Requirement	Resource Requirement After Appending the Default	
	DFT_FROMTYPE set	DFT_FROMTYPE not set
swp>25 && it>10	swp>25 && it>10 && type==$fromtype order[r15s:pg]	swp>25 && it>10 && type==any order [r15s:pg]
ncpus>1 order[ut]	ncpus>1 && type==$fromtype order[ut]	ncpus>1 && type==any order[ut]

Dynamic load information

LSLIB provides several functions to obtain dynamic load information about hosts. dynamic load information is updated periodically by the LIM. The lsInfo data structure returned by the `ls_info(3)` API call stores the definition of all resources. LSF resources are classified into two groups, host-based resources and shared resources. See Administering Platform LSF for more information on host-based and shared resources.

Dynamic host-based resource information

Dynamic host-based resources are frequently referred to as load indices, consisting of 12 built-in load indices and 256 external load indices which can be collected using an ELIM (see *Administering Platform LSF* for more information). The built-in load indices report load information about the CPU, memory, disk subsystem, interactive activities, etc. on each host. The external load indices are optionally defined by your LSF administrator to collect additional host-based dynamic load information for your site.

ls_load()

`ls_load()` reports information about load indices:

```
struct hostLoad *ls_load(resreq, numhosts, options, fromhost)
```

On success, `ls_load()` returns an array containing a hostLoad structure for each host. On failure, it returns `NULL` and sets `lserrno` to indicate the error.

`ls_load()` has the following parameters:

```
char *resreq;   Resource requirements that each host must satisfy
int *numhosts;  Initially contains the number of hosts requested
int options;    Option flags that affect the selection of hosts
char *fromhost; Used in conjunction with the DFT_FROMTYPE option
```

numhosts parameter

*numhosts determines how many hosts should be returned. If *numhosts is 0, information is requested on all hosts satisfying `resreq`. If numhosts is `NULL`, load information is requested on one host. If numhosts is not `NULL`, the number of hostLoad structures returned.

options parameter

The options parameter is constructed from the bitwise inclusive OR of zero or more of the option flags defined in `<lsf/lsf.h>`. The most commonly used flags are:

EXACT

Exactly *numhosts hosts are desired. If `EXACT` is set, either exactly *numhosts hosts are returned, or the call returns an error. If `EXACT` is not set, then up to *numhosts hosts are returned. If *numhosts is `0`, then the `EXACT` flag is ignored and as many eligible hosts in the load sharing system (that is, those that satisfy the resource requirement) are returned.

OK_ONLY

Return only those hosts that are currently in the `ok` state. If `OK_ONLY` is set, hosts that are `busy`, `locked`, `unlicensed`, or `unavail` are not returned. If `OK_ONLY` is not set, then some or all of the hosts whose status are not `ok` may also be returned, depending on the value of *numhosts and whether the `EXACT` flag is set.

NORMALIZE

Normalize CPU load indices. If `NORMALIZE` is set, then the CPU run queue length load indices `r15s`, `r1m`, and `r15m` of each returned host are normalized. See *Administering Platform LSF* for different types of run queue lengths. The default is to return the *raw run queue length*.

EFFECTIVE

If `EFFECTIVE` is set, then the CPU run queue length load indices of each host returned are the effective load. The default is to return the *raw run queue length*. The options `EFFECTIVE` and `NORMALIZE` are mutually exclusive.

IGNORE_RES

Ignore the status of RES when determining the hosts that are considered to be “ok”. If IGNORE_RES is specified, then hosts with RES not running are also considered to be “ok” during host selection.

DFT_FROMTYPE

This flag determines the default resource requirements.

Returns hosts with the same type as the fromhost which satisfy the resource requirements.

fromhost parameter

The fromhost parameter is used when `DFT_FROMTYPE` is set in options. If fromhost is `NULL`, the local host is assumed. `ls_load()` returns an array of the following data structure as defined in `<lsf/lsf.h>`:

```
struct hostLoad {
    char hostName[MAXHOSTNAMELEN];  Name of the host
    int  status[2];            The operational and load status of the host
    float *li;                 Values for all load indices of this host
};
```

The returned hostLoad array is ordered according to the *order requirement* in the resource requirements. For details about the ordering of hosts, see *Administering Platform LSF*.

Example

The following example takes no options, and periodically displays the host name, host status, and 1-minute effective CPU run queue length for each Sun SPARC host in the LSF cluster.

```
/******************************************************
* LSLIB -- Examples
*
* simload
* Displays load information about all Solaris hosts in * the cluster.
******************************************************/
#include <stdio.h>
#include <lsf/lsf.h>
#include <string.h>
#include <stdlib.h>

int main()
{
    int i;
    struct hostLoad *hosts;
    char   *resreq="type==SUNSOL";
    int    numhosts = 0;
    int    options = 0;
    char   *fromhost = NULL;
    char   field[20]="*";
/* get load information on specified hosts */
    hosts = ls_load(resreq, &numhosts, options, fromhost);
    if (hosts == NULL) {
        ls_perror("ls_load");
        exit(-1);
    }
/* print out the host name, host status and the 1-minute CPU run queue length
*/
    printf("%-15.15s %6.6s%6.6s\n", "HOST_NAME", "status",           "r1m");
    for (i = 0; i < numhosts; i++) {
        printf("%-15.15s ", hosts[i].hostName);
        if (LS_ISUNAVAIL(hosts[i].status))
            printf("%6s", "unavail");
        else if (LS_ISBUSY(hosts[i].status))
            printf("%6.6s", "busy");
        else if (LS_ISLOCKED(hosts[i].status))
            printf("%6.6s", "locked");
        else
            printf("%6.6s", "ok");

        if (hosts[i].li[R1M] >= INFINIT_LOAD)
            printf("%6.6s\n", "-");
        else {
            sprintf(field + 1, "%5.1f", hosts[i].li[R1M]);
            if (LS_ISBUSYON(hosts[i].status, R1M))
        printf("%6.6s\n", field);
        else
        printf("%6.6s\n", field + 1);
        }
    }
    exit(0);
}
```

The output of the above program is similar to the following:

```
% a.out
HOST_NAME       status    r1m
hostB             ok      0.0
hostC             ok      1.2
hostA            busy     0.6
hostD            busy     *4.3
hostF            unavail
```

If the host status is `busy` because of `r1m`, then an asterisk (`*`) is printed in front of the value of the `r1m` load index.

In the above example, the returned data structure hostLoad never needs to be freed by the program even if `ls_load()` is called repeatedly.

Each element of the li array is a floating point number between 0.0 and `INFINIT_LOAD` (defined in `lsf.h`). The index value is set to `INFINIT_LOAD` by LSF to indicate an invalid or unknown value for an index.

The li array can be indexed using different ways. The constants defined in `lsf.h` (see the `ls_load(3)` man page) can be used to index any built-in load indices as shown in the above example. If external load indices are to be used, the order in which load indices are returned will be the same as that of the resources returned by `ls_info()`. The variables numUsrIndx and numIndx in structure lsInfo can be used to determine which resources are load indices.

Tip:

There are more flexible ways to map load index names to values.

LSF defines a set of macros in `lsf.h` to test the status field. The most commonly used macros include:

Macro Name	Macro Description
`LS_ISUNAVAIL(status)`	Returns 1 if the LIM on the host is unavailable.
`LS_ISBUSYON(status, index)`	Returns 1 if the host is busy on the given index.
`LS_ISBUSY(status)`	Returns 1 if the host is busy.
`LS_ISLOCKEDU(status)`	Returns 1 if the host is locked by user.
`LS_ISLOCKEDW(status)`	Returns 1 if the host is locked by a time window.
`LS_ISLOCKED(status)`	Returns 1 if the host is locked.
`LS_ISRESDOWN(status)`	Returns 1 if the RES is down.
`LS_ISSBDDOWN(status)`	Returns 1 if the SBATCH is down.
`LS_ISUNLICENSED(status)`	Returns 1 if the host has no software license.
`LS_ISOK(status)`	Returns 1 if none of the above is true.
`LS_ISOKNRES(status)`	Returns 1 if the host is ok except that no RES or SBATCHD is running.

Dynamic shared resource information

Unlike host-based resources which are inherent properties contributing to the making of each host, shared resources are shared among a set of hosts. The availability of a shared resource is characterized by having multiple instances, with each instance being shared among a set of hosts.

ls_sharedresource-info()

`ls_sharedresourceinfo()` can be used to access shared resource information:

```
struct lsSharedResourceInfo *ls_sharedresourceinfo(resources, numResources,
hostname, options)
```

On success, `ls_sharedresourceinfo()` returns an array containing a shared resource information structure (`struct lsSharedResourceInfo`) for each shared resource. On failure, `ls_sharedresourceinfo()` returns `NULL` and sets `lserrno` to indicate the error.

`ls_sharedresourceinfo()` has the following parameters:

```
char **resources;           NULL terminated array of resource names
int *numresources;          Number of shared resources
int hostName;               Host name
int options;                Options (Currently set to 0)
```

resources Parameter

resources is a list (`NULL` terminated array) of shared resource names whose resource information is to be returned. Specify `NULL` to return resource information for all shared resources defined in the cluster.

numresources Parameter

numresources is an integer specifying the number of resource information structures (LS_SHARED_RESOURCE_INFO_T) to return. Specify 0 to return resource information for all shared resources in the cluster. On success, numresources is assigned the number of LS_SHARED_RESOURCE_INFO_T structures returned.

hostName Parameter

hostName is the integer name of a host. Specifying hostName indicates that only the shared resource information for the named host is to be returned. Specify `NULL` to return resource information for all shared resources defined in the cluster.

options Parameter

options is reserved for future use. Currently, it should be set to 0.

lsSharedResource-Info structure

`ls_sharedresourceinfo()` returns an array of the following data structure as defined in `<lsf/lsf.h>`:

```
typedef struct lsSharedResourceInfo {
    char  *resourceName;      Resource name
    int   nInstances;         Number of instances
    LS_SHARED_RESOURCE_INST_T *instances;  Pointer to the next instance
} LS_SHARED_RESOURCE_INFO_T
```

For each shared resource, LS_SHARED_RESOURCE_INFO_T encapsulates an array of instances in the instances field. Each instance is represented by the data type LS_SHARED_RESOURCE_INST_T defined in `<lsf/lsf.h>`:

```
typedef struct lsSharedResourceInstance {
    char  *value;             Value associated with the instance
    int   nHosts;             Number of hosts sharing the instance
    char  **hostList;         Hosts associated with the instance
} LS_SHARED_RESOURCE_INST_T;
```

The value field of the LS_SHARED_RESOURCE_INST_T structure contains the ASCII representation of the actual value of the resource. The interpretation of the value requires the knowledge of the resource (Boolean, Numeric or String), which can be obtained from the resItem structure accessible through the lsLoad structure returned by `ls_load()`.

Example

The following example shows how to use `ls_sharedresourceinfo()` to collect dynamic shared resource information in an LSF cluster. This example displays information from all the dynamic shared resources in the cluster. For each resource, the resource name, instance number, value and locations are displayed.

```
#include <stdio.h>
#include <lsf/lsf.h>
static struct resItem * getResourceDef(char *);
static struct lsInfo  * lsInfo;

void
int main()
{
    struct lsSharedResourceInfo *resLocInfo;
    int numRes = 0;
    int i, j, k;

    lsInfo = ls_info();
    if (lsInfo == NULL) {
        ls_perror("ls_info");
        exit(-1);
    }

    resLocInfo = ls_sharedresourceinfo (NULL, &numRes, NULL, 0);

    if (resLocInfo == NULL) {
        ls_perror("ls_sharedresourceinfo");
        exit(-1);
    }

    printf("%-11.11s %8.8s %6.6s %14.14s\n", "NAME",
          "INSTANCE", "VALUE", "LOCATIONS");

    for (k = 0; k < numRes; k++) {
        struct resItem *resDef;
        resDef = getResourceDef(resLocInfo[k].resourceName);
        if (! (resDef->flags & RESF_DYNAMIC))
            continue;

        printf("%-11.11s", resLocInfo[k].resourceName);
        for (i = 0; i < resLocInfo[k].nInstances; i++) {
            struct lsSharedResourceInstance *instance;

            if (i == 0)
                printf(" %8.1d", i+1);
            else
                printf(" %19.1d", i+1);

            instance = &resLocInfo[k].instances[i];
            printf(" %6.6s", instance->value);

            for (j = 0; j < instance->nHosts; j++)
                if (j == 0)
                   printf(" %14.14s\n", instance->hostList[j]);
                else
                   printf(" %41.41s\n", instance->hostList[j]);

        } /* for */
    } /* for */
} /* main */
static struct resItem *
getResourceDef(char *resourceName)
{
    int i;

    for (i = 0; i < lsInfo->nRes; i++) {
        if (strcmp(resourceName, lsInfo->resTable[i].name) == 0)
            return &lsInfo->resTable[i];
    }

    /* Fail to find the matching resource */
    fprintf(stderr, "Cannot find resource definition for           <%s>\n",
resourceName);

    exit (-1);

}
```

The output of the above program is similar to the following:

```
% a.out
NAME        INSTANCE  VALUE    LOCATIONS
dynamic1        1       2        hostA
                                 hostC
                                 hostD
                2       4        hostB
                                 hostE
dynamic2        1       3        hostA
                                 hostE
```

The resource dynamic1 has two instances, one contains two resource units shared by `hostA`, `hostC` and `hostD` and the other contains four resource units shared by `hostB` and `hostE`.

The dynamic2 resource has only one instance with three resource units shared by `hostA` and `hostE`.

For configuration of shared resources, see the ResourceMap section of `lsf.cluster.cluster_name` file in the *Platform LSF Reference*.

Placement decisions

If you are writing an application that needs to run tasks on the best available hosts, you need to make a *placement decision* as to which task each host should run.

Placement decisions take the resource requirements of the task into consideration. Every task has a set of resource requirements. These may be static, such as a particular hardware architecture or operating system, or dynamic, such as an amount of swap space for virtual memory.

LSLIB provides services for placement advice. All you have to do is to call the appropriate LSLIB function with appropriate resource requirements.

A placement advice can be obtained by calling either the `ls_load()` function or the `ls_placereq()` function. `ls_load()` returns a placement advice together with load index values. `ls_placereq()` returns only the qualified host names. The result list of hosts are ordered by preference, with the first being the best. `ls_placereq()` is useful when a simple placement decision would suffice. `ls_load()` can be used if the placement advice from LSF must be adjusted by your additional criteria. The LSF utilities `lsrun`, `lsmake`, `lslogin`, and `lstcsh` all use `ls_placereq()` for placement decision. lsbatch, on the other hand, uses `ls_load()` to get an ordered list of qualified hosts, and then makes placement decisions by considering lsbatch-specific policies.

In order to make optimal placement decisions, it is important that your resource requirements best describe the resource needs of the application. For example, if your task is memory intensive, then your resource requirement string should have ‘mem’ in the order segment, ‘fddi order[mem:r1m]’.

ls_placereq()

`ls_placereq()` takes the form of:

```
char **ls_placereq(resreq, num, options, fromhost)
```

On success, `ls_placereq()` returns an array of host names that best meet the resource requirements. Hosts listings may be duplicated for hosts that have sufficient resources to accept multiple tasks (for example, multiprocessors).

On failure, `ls_placereq()` returns `NULL` and sets `lserrno` to indicate the error.

The parameters for `ls_placereq()` are very similar to those of the `ls_load()` function described in the previous section.

LSLIB will append default resource requirement to `resreq` according to the rules described in “Handling Default Resource Requirements”.

Preference is given to fromhost over remote hosts that do not have a significantly lighter load or greater resources. This preference avoids unnecessary task transfer and reduces overhead. If fromhost is `NULL`, then the local host is assumed.

Example

The following example takes a resource requirement string as an argument and displays the host in the LSF cluster that best satisfies the resource requirement.

```
#include <stdio.h>
#include <lsf/lsf.h>

main(argc, argv)
    int  argc;
    char *argv[];
{
    char *resreq = argv[1];
    char **best;
    int  num = 1;
    int  options = 0;
    char *fromhost = NULL;

/* check the input format */

    if (argc != 2 ) {
        fprintf(stderr, "Usage: %s resreq\n", argv[0]);
        exit(-2);
    }

/* find the best host with the given condition (e.g. resource requirement) */

    best = ls_placereq(resreq, &num, options, fromhost);
    if (best == NULL) {
        ls_perror("ls_placereq()");
        exit(-1);
    }
    printf("The best host is <%s>\n", best[0]);

    exit(0);
}
```

The above program will produce output similar to the following:

```
% a.out "type==local order[r1m:ls]"
The best host is <hostD>
```

LSLIB also provides a variant of `ls_placereq()`. `ls_placeofhosts()` lets you provide a list of candidate hosts. See the `ls_policy(3)` man page for details.

Task resource requirements

Host selection relies on resource requirements. To avoid the need to specify resource requirements each time you execute a task, LSF maintains a list of task names together with their default resource requirements for each user. This information is kept in three task list files: the system-wide defaults, the per-cluster defaults, and the per-user defaults.

A user can put a task name together with its resource requirements into his/her remote task list by running the `lsrtasks` command. The `lsrtasks` command can be used to add, delete, modify, or display a task entry in the task list. For more information on remote task list and an explanation of resource requirement strings, see Administering Platform LSF.

ls_resreq()

`ls_resreq()` gets the resource requirements associated with a task name. With `ls_resreq()`, LSF applications or utilities can automatically retrieve the resource requirements of a given task if the user does not explicitly specify it. For example, the LSF utility `lsrun` tries to find the resource requirements of the user-typed command automatically if '-R' option is not specified by the user on the command line.

The syntax of `ls_resreq()` is:

```
char *ls_resreq(taskname)
```

If taskname does not appear in the remote task list, `ls_resreq()` returns `NULL`.

Typically the resource requirements of a task are then used for host selection purpose. The following program takes the input argument as a task name, get the associated resource requirements from the remote task list, and then supply the resource requirements to a `ls_placereq()` call to get the best host for running this task.

Example

```
#include <stdio.h>
#include <lsf/lsf.h>

int main(int argc, char *argv[])

{
    char *taskname = argv[1];
    char *resreq;
    char **best;

/* check the input format */

    if (argc != 2 ) {
        fprintf(stderr, "Usage: %s taskname\n", argv[0]);
        exit(-1);
    }

    resreq = ls_resreq(taskname);

/* get the resource requirement for the given command */

    if (resreq)
       printf("Resource requirement for %s is \"%s\".\n",
             taskname, resreq);
    else
       printf("Resource requirement for %s is NULL.\n", taskname);

/* select the best host with the given resource requirement to run the job */

    best = ls_placereq(resreq, NULL, 0, NULL);
    if (best == NULL) {
        ls_perror("ls_placereq");
        exit(-1);
    }
    printf("Best host for %s is <%s>\n", taskname, best[0]);

    exit(0);
}
```

The above program will produce output similar to the following:

```
% a.out myjob
Resource requirement for myjob is "swp>50 order[cpu:mem]"
Best host for myjob is <hostD>
```

Remote execution services

Remote execution of interactive tasks in LSF is supported through the Remote Execution Server (RES). The RES listens on a well-known port for service requests. Applications initiate remote execution by making an LSLIB call.

Initialize an application for remote execution

Before executing a task remotely, an application must call the `ls_initrex()`:

```
int ls_initrex(numports, options)
```

ls_initrex()

On success, `ls_initrex()` initializes the LSLIB for remote execution. If your application is installed as a setuid program, `ls_initrex()` returns the number of socket descriptors bound to privileged ports. If your program is not installed as a setuid to root program, `ls_initrex()` returns numports on success.

On failure, `ls_initrex()` returns -1 and sets the global variable `lserrno` to indicate the error.

Tip:

`ls_initrex()` must be called before any other remote execution function (see `ls_rex(3)`) or any remote file operation function (see `ls_rfs(3)`) in LSLIB can be called.

`ls_initrex()` has the following parameters:

```
int numports;    The number of priviliged ports to create
int options;      Either KEEPUID or 0
```

If your program is installed as a setuid to root program, numports file descriptors, starting from FIRST_RES_SOCK (defined in `<lsf/lsf.h>`), are bound to privileged ports by `ls_initrex()`. These sockets are used only for remote connections to RES. If numports is 0, then the system will use the default value LSF_DEFAULT_SOCKS defined in `lsf.h`.

By default, `ls_initrex()` restores the effective user ID to real user ID if the program is installed as a setuid to root program. If options is set to `KEEPUID` (defined in `lsf.h`), `ls_initrex()` preserves the current effective user ID. This option is useful if the application needs to be a setuid to root program for some other purpose as well and does not want to go back to real user ID immediately after `ls_initrex()`.

Caution:

If KEEPUID flag is set in options, you must make sure that your application restores back to the real user ID at a proper time of the program execution.

`ls_initrex()` function selects the security option according to the following rule: if the application program invoking it has an effective uid of root, then privileged ports are created. If there are no privileged port created and, at remote task start-up time, RES will use the authentication protocol defined by `LSF_AUTH` in the `lsf.conf` file.

Run a task remotely

The example program below runs a command on one of the best available hosts. It makes use of:

- `ls_resreq()`
- `ls_placereq()`
- `ls_initrex()`
- `ls_rexecv()`:

```
int ls_rexecv(host, argv, options)
```

`ls_rexecv()` executes a program on the specified host. It does not return if successful. It returns -1 on failure.

`ls_rexecv()` is like a remote `execvp`. If a connection with the RES on *a host* has not been established, `ls_rexecv()` sets one up. The remote execution environment is set up to be exactly the same as the local one and is cached by the remote RES server. `ls_rexecv()` has the following parameters:

```
char *host;                 The execution host
char *argv[];               The command and its arguments
int options;                 See below
```

The options argument is constructed from the bitwise inclusive OR of zero or more or the option flags defined in `<lsf/lsf.h>` with names starting with ‘REXF_’. the group of flags are as follows:

REXF_USEPTY

Use a remote pseudo terminal as the stdin, stdout, and stderr of the remote task. This option provides a higher degree of terminal I/O transparency. This is needed only when executing interactive screen applications such as `vi`. The use of a pseudo-terminal incurs more overhead and should be used only if necessary. This is the most commonly used flag.

REXF_CLNTDIR

Use the local client’s current working directory as the current working directory for remote execution.

REXF_TASKPORT

Request the remote RES to create a task port and return its number to the LSLIB.

REXF_SHMODE

Enable shell mode support if the REXF_USEPTY flag is also given. This flag is ignored if REXF_USEPTY is not given. This flag should be specified for submitting interactive shells, or applications which redefine, or applications which redefine the ctrl-C and ctrl-Z keys (e.g. `jove`).

LSLIB also provides `ls_rexecve()` to specify the environment to be set up on the remote host.

Example

The program follows:

```
#include <stdio.h>
#include <lsf/lsf.h>
```

```
main(argc, argv)
    int  argc;
    char *argv[];
{
    char *command;
    char *resreq;
    char **best;
    int  num = 1;

/* check the input format */
    if (argc < 2 ) {
       fprintf(stderr, "Usage: %s command [argument ...]\n",
              argv[0]);
        exit(-1);
    }

command = argv[1];

/* initialize the remote execution */
    if (ls_initrex(1, 0) < 0) {
        ls_perror("ls_initrex");
        exit(-1);
    }

/* get resource requirement for the given command */
    resreq = ls_resreq(command);

    best = ls_placereq(resreq, &num, 0, NULL);
    if (best == NULL) {
        ls_perror("ls_placereq()");
        exit(-1);
    }

/* start remote execution on the selected host for the job */
    printf("<<Execute %s on %s>>\n", command, best[0]);
    ls_rexecv(best[0], argv + 1, 0);

    /* if the remote execution is successful, the following lines will not be
executed */
    ls_perror("ls_rexecv()");
    exit(-1);
}
```

The output of the above program would be something like:

```
% a.out myjob
<<Execute myjob on hostD>>
(output from myjob goes here ....)
```

Tip:

Any application that uses LSF’s remote execution service must be installed for proper authentication.

The LSF command `lsrun` is implemented using the `ls_rexecv()` function. After remote task is initiated, `lsrun` calls the `ls_rexecv()` function, which then executes NIOS to handle all input/output to and from the remote task and exits with the same status when remote task exits.

CHAPTER

3

Programming wth LSBLIB

About LSBLIB

Since LSF batch is built on top of LSF base, LSBLIB relies on services provided by LSLIB. However, you only need to link your program with LSBLIB to use LSBLIB functions because the header file of LSBLIB (`lsbatch.h`) already includes the LSLIB (`lsf.h`). All other LSF products (such as Platform Parallel and Platform Make) relies on services provided by LSBLIB.

LSF batch and Platform JobScheduler services are provided by `mbatchd`. Services for processing event and job log files which do not involve any daemons. LSBLIB is shared by both LSF batch and Platform JobScheduler. The functions described for LSF batch in this chapter also apply to other LSF products, unless explicitly indicated otherwise.

LSF batch applications

Before accessing any of the LSF batch services, an application must initialize LSBLIB. An application does this by calling `lsb_init()`.

lsb_init() function

`lsb_init()` has the following parameter:

```
char *appName
```

On success, `lsb_init()` returns 0. On failure, it returns -1 and sets `lsberrno` to indicate the error.

The parameter appName is the name of the application. Use appName to log detailed messages about the transactions inside LSLIB for debugging purpose. If `LSB_CMD_LOG_MASK` is defined as `LOG_DEBUG1`, the messages will be logged.

Messages are logged in `LSF_LOGDIR/appname`. If appname is `NULL`, the log file is `LSF_LOGDIR/bcmd`.

Example

Here is an example of code showing the usage of this function:

```
/* Include <lsf/lsbatch.h> when using this function */
if (lsb_init(argv[0]) < 0) {
        lsb_perror("simbsub: lsb_init() failed");
        exit(-1);
}
```

lsb_perror()

The function `lsb_perror(char *usrMsg)` prints a batch LSF error message on `stderr`. The user message usrMsg is printed, followed by a colon (`:`) and the batch error message corresponding to `lsberrno`.

LSF batch queues

LSF batch queues hold jobs in LSF batch and according to scheduling policies and limits on resource usage.

lsb_queueinfo()

`lsb_queueinfo()` gets information about the queues in LSF batch. This includes:

- Queue name
- Parameters
- Statistics
- Status
- Resource limits
- Scheduling policies and parameters
- Users and hosts associated with the queue.

The example program in this section uses `lsb_queueinfo()` to get the queue information:

```
struct queueInfoEnt *lsb_queueinfo(queues,numQueues,
                   hostname,username,options)
```

`lsb_queueinfo()` has the following parameters:

```
char  **queues;            Array containing names of queues of interest
int   *numQueues;          Number of queues
char  *hostname;           Specified queues using hostname
char  *username;           Specified queues enabled for user
int   options;             Reserved for future use; supply 0
```

To get information on all queues, set *numQueues to 0. If *numQueues is 1 and queue is `NULL`, information on the default system queue is returned.

If hostname is not `NULL`, then all queues using host hostname as a batch server host will be returned. If username is not `NULL`, then all queues allowing user username to submit jobs to will be returned.

On success, `lsb_queueinfo()` returns an array containing a queueInfoEnt structure (see below) for each queue of interest and sets *numQueues to the size of the array. On failure, `lsb_queueinfo()` returns `NULL` and sets `lsberrno` to indicate the error.

The queueInfoEnt structure is defined in `lsbatch.h` as

```
struct queueInfoEnt {
    char  *queue;            Name of the queue
    char  *description;      Description of the queue
    int   priority;          Priority of the queue
    short nice;              Value that runs jobs in the queue
    char  *userList;         Users allowed to submit jobs to the queue
    char  *hostList;         Hosts that can run jobs in the queue
    int   nIdx;              Size of the loadSched and loadStop arrays
    float *loadSched;        Load thresholds that control scheduling of job
                                 from the queue
    float *loadStop;          Load thresholds that control suspension of
                                 jobs from the queue
    int   userJobLimit;      Number of unfinished jobs a user can dispatch
                                 from the queue
    int   procJobLimit;      Number of unfinished jobs the queue can
                                 dispatch to a processor
    char  *windows;          Queue run window
    int   rLimits[LSF_RLIM_NLIMITS];  Per-process resource limits for
                                          jobs
    char  *hostSpec;         Obsolete. Use defaultHostSpec instead
    int   qAttrib;           Attributes of the queue
    int   qStatus;           Status of the queue
    int   maxJobs;           Job slot limit of the queue.
    int   numJobs;            Total number of job slots required by all jobs
    int   numPEND;            Number of job slots needed by pending jobs
    int   numRUN;             Number of jobs slots used by running jobs
    int   numSSUSP;           Number of job slots used by system
                                  suspended jobs
    int   numUSUSP;           Number of jobs slots used by user
                                  suspended jobs
    int   mig;                Queue migration threshold in minutes
    int   schedDelay;         Schedule delay for new jobs
    int   acceptIntvl;        Minimum interval between two jobs dispatched
                                  to the same host
    char  *windowsD;          Queue dispatch window
    char  *nqsQueues;         Blank-separated list of NQS queue specifiers
    char  *userShares;        Blank-separated list of user shares
    char  *defaultHostSpec;   Value of DEFAULT_HOST_SPEC for the
                                  queue in lsb.queues
    int   procLimit;          Maximum number of job slots a job can take
    char  *admins;            Queue level administrators
    char  *preCmd;            Queue level pre-exec command
    char  *postCmd;           Queue’s post-exec command
    char  *requeueEValues;    Queue’s requeue exit status
    int   hostJobLimit;       Per host job slot limit
    char  *resReq;            Queue level resource requirement
    int   numRESERVE;         Reserved job slots for pending jobs
    int   slotHoldTime;       Time period for reserving job slots
    char  *sndJobsTo;         Remote queues to forward jobs to
    char  *rcvJobsFrom;       Remote queues which can forward to me
    char  *resumeCond;        Conditions to resume jobs
    char  *stopCond;          Conditions to suspend jobs
    char  *jobStarter;        Queue level job starter
    char  *suspendActCmd;     Action commands for SUSPEND
    char  *resumeActCmd;      Action commands for RESUME
    char  *terminateActCmd;   Action commands for TERMINATE
    int   sigMap[LSB_SIG_NUM];  Configurable signal mapping
    char  *preemption;        Preemption policy
    int    maxRschedTime;     Time period for remote cluster to schedule job
    struct shareAcctInfoEnt *shareAccts;  Array of shareAcctInfoEnt
    char   *chkpntDir;        chkpnt directory
    int    chkpntPeriod;      chkpnt period
    int    imptJobBklg;       Number of important jobs kept in the queue
    int    defLimits[LSF_RLIM_NLIMITS];  LSF resource limits (soft)
    int    chunkJobSize;      Maximum number of jobs in one chunk
    int    minProcLimit;      Minimum processor limit
    int    defProcLimit;      Default processor limit
    char   *fairshareQueues;
    char   *defExtSched;      Default external scheduling
    char   *mandExtSched;     Mandatory external scheduling
    int    slotShare;         The share of cpus to use in the pool
    char   *slotPool;         The cpu pool name
    int    underRCond;
    int    overRCond;
    float  idleCond;
    int    underRJobs;
    int    overRJobs;
    int    idleJobs;
    int    warningTimePeriod;  Warning time period in seconds
    char   *warningAction;     Warning action, SIGNAL | CHKPNT | command */
    char   *qCtrlMsg;          AdminAction - queue control message*/
    char   *acResReq;
    int   symJobLimit;         Limit of running service job/symphony job*/
    char   *cpuReq;            cpu_req for service partition of symphony */
    int    proAttr;          Indicates willingness to donate/borrow
    int    lendLimit;        Grace period to lend/return idle hosts
   int    hostReallocInterval;  Grace period to lend/return idle hosts
   int    numCPURequired;   Number of cpus required by CPU provision
    int    numCPUAllocated;  Number of cpus actually allocated
    int    numCPUBorrowed;   Number of cpus borrowed
    int    numCPULent;       Number of cpus lent
    /* the number of reserved cpu(numCPUReserved) = numCPUAllocated - numCPUBorrowed + numCPULent
*/
/* the following fields are for real-time app(ex. murex) of symphony */
    int    schGranularity;        Scheduling granularity in milliseconds
    int    symTaskGracePeriod;    Grace period for stopping symphony tasks
    int    minOfSsm;              Minimum number of ssm
    int    maxOfSsm;              Maximum number of ssm
    int    numOfAllocSlots;       Number of allocated slots
    char *servicePreemption;      Service preemptin policy
   int    provisionStatus;       Dynamic cpu provision status
    int    minTimeSlice;          Minimal time for preemt. backfill (sec)
    char   *queueGroup;           List of queues defined in QUEUE_GROUP
    int    numApsFactors;
    struct apsFactorInfo *apsFactorInfoList;
    struct apsFactorMap  *apsFactorMaps;  Mapping from factors to subfactors
    struct apsLongNameMap *apsLongNames;  Mapping from factors to their long names
    int    maxJobPreempt;     Maximum number of job preempted times
    int    maxPreExecRetry;   Maximum number of pre-exec retry times
    int    localMaxPreExecRetry;   Maximum number of pre-exec retry times for local cluster
    int    maxJobRequeue;     Maximum number of job re-queue times
    int    usePam;            Use Linux-PAM
    int    cu_type_exclusive; Compute unit type
    char  *cu_str_exclusive;  String specified in EXCLUSIVE=CU[<string>]
};
```

The variable nIdx is the number of load threshold values for job scheduling. This is the total number of load indices returned by LIM. The parameters sndJobsTo, rcvJobsFrom, and maxRschedTime are used with LSF MultiCluster. The variable chunkJobSize must be larger than 1.

For a complete description of the fields in the queueInfoEnt structure, see the `lsb_queueinfo()` man page.

Include `lsbatch.h` in every application that uses LSBLIB functions. `lsf.h` does not have to be explicitly included in your program because `lsbatch.h` includes `lsf.h`.

Like the data structures returned by LSLIB functions, the data structures returned by an LSBLIB function are dynamically allocated inside LSBLIB and are automatically freed next time the same function is called. Do not attempt to free the space allocated by LSBLIB. To keep this information across calls, make your own copy of the data structure.

Example

The program below takes a queue name as the first argument and displays information about the named queue.

```
/******************************************************
* LSBLIB -- Examples
*
* simbqueues
* Display information about a specific queue in the
* cluster.
* (Queue name is given on the command line argument)
* It is similar to the command "bqueues QUEUE_NAME".
******************************************************/
# include <lsf/lsbatch.h>
int main (int argc, char *argv[])
{
    struct queueInfoEnt *qInfo;
    char *queues;
        /* take the command line argument as the queue name */
    int numQueues = 1;
        /* only 1 queue name in the array queue */
    char *host = NULL;/* all queues are of interest */
    char *user = NULL;/* all queues are of interest */
    int options = 0;
    /* check if input is in the right format: "./simbqueues
    QUEUENAME" */
    if (argc != 2) {
        printf("Usage: %s queue_name\n", argv[0]);
        exit(-1);
    }
    queues = argv[1];
/* initialize LSBLIB and get the configuration environment */
    if (lsb_init(argv[0]) < 0) {
        lsb_perror("simbqueues: lsb_init() failed");
        exit(-1);
    }
    /* get queue information about the specified queue */
    qInfo = lsb_queueinfo(&queues, &numQueues, host, user, 
    options);
    if (qInfo == NULL) {
        lsb_perror("simbqueues: lsb_queueinfo() failed");
        exit(-1);
    }
    /* display the queue information (name, descriptions, 
    priority, nice value, max num of jobs, num of PEND, RUN, 
    SUSP and TOTAL jobs) */
    printf("Information about %s queue:\n", queues);
    printf("Description: %s\n", qInfo[0].description);
    printf("Priority: %d     Nice: %d     \n",            qInfo[0].priority, qInfo[0].nice);
    printf("Maximum number of job slots:");
    if (qInfo->maxJobs < INFINIT_INT)
        printf("%5d\n", qInfo[0].maxJobs);
    else
        printf("%5s\n", "unlimited");
    printf("Job slot statistics: PEND(%d) RUN(%d) SUSP(%d)            TOTAL(%d).\n", qInfo
[0].numPEND, qInfo[0].numRUN,            qInfo[0].numSSUSP + qInfo[0].numUSUSP,            qInfo
[0].numJobs);
    exit(0);
} /* main */
```

In the above program, `INFINIT_INT` is defined in `lsf.h` and is used to indicate that there is no limit set for maxJobs. This applies to all Platform LSF API function calls. Platform LSF will supply `INFINIT_INT` automatically whenever the value for the variable is either invalid (not available) or infinity. This value should be checked for all variables that are optional. For example, if you display the loadSched/loadStop values, an `INFINIT_INT` indicates that the threshold is not configured and is ignored.

Similarly, `lsb_perror()` prints error messages regarding function call failure. You can check `lsberrno` if you want to take different actions for different errors.

The above program will produce output similar to the following:

```
Information about normal queue:
Description: For normal low priority jobs
Priority: 25            Nice: 20
Maximum number of job slots : 40
Job slot statistics: PEND( 5) RUN(12) SUSP(1) TOTAL(18)
```

LSF batch hosts

LSF batch execution hosts execute jobs in the LSF batch system.

lsb_hostinfo()

LSBLIB provides `lsb_hostinfo()` to get information about the server hosts in LSF batch. This includes configured static and dynamic information. Examples of host information include: host name, status, job limits and statistics, dispatch windows, and scheduling parameters.

The example program in this section uses `lsb_hostinfo()`:

```
struct hostInfoEnt *lsb_hostinfo(hosts, numHosts)
```

`lsb_hostinfo()` gets information about LSF batch server hosts. On success, it returns an array of hostInfoEnt structures which hold the host information and sets *numHosts to the size of the array. On failure, `lsb_hostinfo()` returns `NULL` and sets `lsberrno` to indicate the error.

`lsb_hostinfo()` has the following parameters:

```
char  **hosts;                Array of names of hosts of interest
int   *numHosts;               Number of names in hosts
```

To get information on all hosts, set *numHosts to 0. This sets *numHosts to the actual number of hostInfoEnt structures when `lsb_hostinfo()` returns successfully.

If *numHosts is 1 and hosts is `NULL`, `lsb_hostinfo()`returns information on the local host.

hostInfoEnt structure

The hostInfoEnt structure is defined in `lsbatch.h` as

```
struct hostInfoEnt {
    char   *host;
    int    hStatus;          Host status
    int    *busySched;       Host loadSched busy reason
    int    *busyStop;        Host loadStop  busy reason
    float  cpuFactor;
    int    nIdx;             Number of load index
    float  *load;            Load for scheduling batch jobs
    float  *loadSched;       Stop scheduling new jobs if over
    float  *loadStop;        Stop jobs if over this load
    char   *windows;         ASCII desp of run windows
    int    userJobLimit;     Number of jobs per user allowed to run
    int    maxJobs;          Maximum number of jobs allowed to run
    int    numJobs;          Number of total jobs
    int    numRUN;           Number of running jobs
    int    numSSUSP;         Number of system suspended jobs
    int    numUSUSP;         Number of user suspended jobs
    int    mig;   Number of minutes suspended before migration
    int    attr;            Host attributes
#define H_ATTR_CHKPNTABLE  0x1
#define H_ATTR_CHKPNT_COPY 0x2
   float *realLoad;         Effective load of the host
    int   numRESERVE;        Number of slots reserved for pending jobs
   int   chkSig;          If attr has an H_ATTR_CHKPNT_COPY attribute. chkSig is set to the signal
which triggers checkpoint and copy operation. Otherwise, chkSig is set to the signal which triggers
checkpoint operation on the host
    float   cnsmrUsage;    Number of resources used by consumer
    float   prvdrUsage;    Number of resource used by provider
    float   cnsmrAvail;    Number of resources available for consumer
    float   prvdrAvail;    Number of resources available for provider
    float   maxAvail;      Maximum number of resources available
    float   maxExitRate;   Job exit rate threshold on the host
    float   numExitRate;    Number of job exit rate on the host
    char    *hCtrlMsg;     AdminAction - host control message
};
```

There are differences between the host information returned by `ls_gethostinfo()` and the host information returned by the `lsb_hostinfo()`. `ls_gethostinfo()` returns general information about the hosts whereas `lsb_hostinfo()`returns LSF batch specific information about hosts.

For a complete description of the fields in the hostInfoEnt structure, see the `lsb_hostinfo(3)` man page.

Example

The following example takes a host name as an argument and displays information about the named host. It is a simplified version of the LSF batch `bhosts` command.

```
/******************************************************
* LSBLIB -- Examples
*
* simbhosts
* Display information about the batch server host with
* the given name in the cluster.
******************************************************/
#include <lsf/lsbatch.h>
int main (int argc, char *argv[])
{
    struct hostInfoEnt *hInfo;
        /* array holding all job info entries */
    char *hostname = argv[1]; /* given host name */
    int numHosts = 1;/* number of interested host */
    /* check if input is in the right format: "./simbhosts
    HOSTNAME" */
    if (argc!=2) {
        printf("Usage: %s hostname\n", argv[1]);
        exit(-1);
    }
    /* initialize LSBLIB and get the configuration environment */
    if (lsb_init(argv[0]) < 0) {
        lsb_perror("simbhosts: lsb_init() failed");
        exit(-1);
    }
    hInfo = lsb_hostinfo(&hostname, &numHosts);
        /* get host info */
    if (hInfo == NULL) {
        lsb_perror("simbhosts: lsb_hostinfo() failed");
        exit (-1);
    }
    /* display the host information (name,status, job limit,
    num of RUN/SSUSP/USUSP jobs)*/
    printf("HOST_NAME          STATUS    JL/U  NJOBS  RUN
    SSUSP USUSP\n");
    printf ("%-18.18s", hInfo->host);
    if (hInfo->hStatus & HOST_STAT_UNLICENSED)
        printf(" %-9s\n", "unlicensed");
    else if (hInfo->hStatus & HOST_STAT_UNAVAIL)
        printf(" %-9s",  "unavail");
    else if (hInfo->hStatus & HOST_STAT_UNREACH)
        printf(" %-9s", "unreach");
    else if (hInfo->hStatus & ( HOST_STAT_BUSY | HOST_STAT_WIND |
HOST_STAT_DISABLED |
                              HOST_STAT_LOCKED |
                              HOST_STAT_FULL |
                              HOST_STAT_NO_LIM))
        printf(" %-9s", "closed");
    else
        printf(" %-9s", "ok");
    if (hInfo->userJobLimit < INFINIT_INT)
        printf("%4d", hInfo->userJobLimit);
    else
        printf("%4s", "-");
    printf("%7d  %4d  %4d  %4d\n", hInfo->numJobs, hInfo->            numRUN, hInfo->numSSUSP,
hInfo->numUSUSP);
exit(0);
} /* main */
```

The example output from the above program follows:

```
% a.out hostB
HOST_NAME    STATUS    JL/U  NJOBS  RUN  SSUSP USUSP
hostB           ok        -      2     1     1     0
```

hStatus is the status of the host. It is the bitwise inclusive OR of some of the following constants defined in `lsbatch.h`:

Host Status Name	**Host Status Description**
`HOST_STAT_BUSY`	The host load is greater than a scheduling threshold. In this status, no new batch job is scheduled to run on this host.
HOST_STAT_WIND	The host dispatch window is closed. In this status, no new batch job is accepted.
`HOST_STAT_DISABLED`	The host has been disabled by the Platform LSF administrator and will not accept jobs. In this status, no new batch job will be scheduled to run on this host.
`HOST_STAT_LOCKED`	The host is locked by the LSF administrator. In this status, no new batch job is scheduled to run on this host.
`HOST_STAT_FULL`	The host has reached its job limit. In this status, no new batch job is scheduled to run on this host.
`HOST_STAT_UNREACH`	The `sbatchd` on this host is unreachable.
`HOST_STAT_UNAVAIL`	The LIM and `sbatchd` on this host are unreachable.
`HOST_STAT_UNLICENSED`	The host does not have an LSF license.
`HOST_STAT_NO_LIM`	The host is running an `sbatchd` but not a LIM.
`HOST_STAT_EXCLUSIVE`	The host is locked by an exclusive job. In this status, no new batch job is scheduled to run on this host.
`HOST_STAT_LOCKED_MASTER`	The host is locked by the master LIM.
`HOST_STAT_REMOTE_DISABLED`	The remote leased host is disabled by the Platform LSF administrator and will not accept new jobs. This status is used with `HOST_STATUS_LOCKED`.
`HOST_STAT_LEASE_INACTIVE`	The remote host is closed while the lease is renewed or terminated.
`HOST_STAT_DISABLED_RES`	The host is closed because RES is unavailable. This status occurs only when `LSF_HPC_EXTENSIONS="LSB_HCLOSE_BY_RES"` is set in `lsf.conf.`
`HOST_STAT_DISABLED_RMS`	The host is closed because RES is unavailable.
`HOST_STAT_LOCKED_BY_EGO`	The host is locked by EGO.
`HOST_STAT_CLOSED_BY_ADMIN`	The host is closed by the Platform_LSF administrator.
`HOST_STAT_CU_EXCLUSIVE`	The host is locked by a compute unit exclusive job. In this status, no new batch job is scheduled to run on this host.

If none of the above holds, hStatus is set to `HOST_STAT_OK` to indicate that the host is ready to accept and run jobs.

The constant `INFINIT_INT` defined in `lsf.h` is used to indicate that there is no limit set for `userJobLimit`.

Job submission and modification

Job submission and modification are the most common operations in LSF batch. A user can submit jobs to the system and then modify them if the job has not been started.

lsb_submit()

LSBLIB provides `lsb_submit()` for job submission and `lsb_modify()` for job modification.

```
LS_LONG_INT lsb_submit(jobSubReq, jobSubReply)
LS_LONG_INT lsb_modify(jobSubReq, jobSubReply, jobId)
```

On success, these calls return the job ID. On failure, it returns -1, and `lsberrno` set to indicate the error. `lsb_submit()` is similar to `lsb_modify()`, except `lsb_modify()` modifies the parameters of an already submitted job.

Both of these functions use the same data structure:

```
struct submit      *jobSubReq;      Job specifications
struct submitReply *jobSubReply;    Results of job submission
LS_LONG_INT   jobId;                ID of the job to modify (lsb_modify()
                                     only)
```

submit structure

The submit structure is defined in `lsbatch.h` as:

```
struct submit {
    int    options;           Indicates which optional fields are present
    int    options2;          Indicates which additional fields are present
    char   *jobName;          Job name (optional)
    char   *queue;            Submit the job to this queue (optional)
    int    numAskedHosts;     Size of askedHosts (optional)
    char   **askedHosts;      Array of names of candidate hosts (optional)
    char   *resReq;           Resource requirements of the job (optional)
    int    rlimits[LSF_RLIM_NLIMITS];
                              Limits on system resource use by all of the
                                  job’s processes
    char   *hostSpec;         Host model used for scaling rlimits (optional)
    int    numProcessors;     Initial number of processors needed by the job
    char   *dependCond;       Job dependency condition (optional)
    char   *timeEvent         Time event string for scheduled repetitive jobs
                                  (optional)
    time_t beginTime;         Dispatch the job on or after beginTime
    time_t termTime;          Job termination deadline
    int    sigValue;          This variable is obsolete)
    char   *inFile;           Path name of the job’s standard input file
                                  (optional)
    char   *outFile;          Path name of the job’s standard output file
                                 (optional)
    char   *errFile;        Path name of the job’s standard error output file
                                 (optional)
    char   *command;         Command line of the job
    char   *newCommand       New command for bmod (optional)
    time_t chkpntPeriod;     Job is checkpointable with this period (optional)
    char   *chkpntDir;      Directory for this job’s chk directory (optional)
    int    nxf;              Size of xf (optional)
    struct xFile *xf;        Array of file transfer specifications (optional)
    char   *preExecCmd;      Job’s pre-execution command (optional)
    char   *mailUser;        User E-mail address to which the job’s output
                                 are mailed (optional)
    int    delOptions;       Bits to be removed from options
                                 (lsb_modify() only)
    char   *projectName;     Name of the job’s project (optional)
    int    maxNumProcessors;  Requested maximum num of job slots for the
                                  job
    char   *loginShell;      Login shell to be used to re-initialize
                                 environment
    char   *userGroup;       User group
    char   *exceptList;      List of exception handlers
    int    userPriority;     User priority
    char   *rsvId;           Use hosts reserved in advance
    char   *jobGroup;        Job group under which the job runs
    char   *sla;             SLA under which the job runs
    char   *extsched;        extsched options
   int    warningTimePeriod;  Warning time period (seconds), -1 if unspecified
    char   *warningAction;   Warning action, SIGNAL | CHKPNT | command, NULL
if unspecified
    char   *licenseProject;  The license scheduler project
    int    options3;         Extend options again
    int    delOptions3;      Delete options in options3 field
    char   *app;             Application profile
    int  jsdlFlag;           -1 if no -jsdl, and -jsdl_strict options
                            * 0 -jsdl_strict option
                            * 1 -jsdl option*/
    char *jsdlDoc;           jsdl filename*/
    void   *correlator;      ARM correlator */
    char *apsString;   aps string set by admin to denote system value
                           * or admin factor value
    char  *postExecCmd;     Post-execution commands specified by -Ep
    char   *cwd;            CWD specified by -cwd
    int      runtimeEstimation;  Runtime estimation specified by -We
    char *requeueEValues; /* -Q: Job level requeue exit values */
    int     initChkpntPeriod; Initial checkpoint period */
    int     migThreshold;     Migration threshold */
    char *notifyCmd;  Script or command invoked when resize request satisfied
};
```

For a complete description of the fields in the submit structure, see the `lsb_submit(3)` man page.

submitReply structure

The submitReply structure is defined in `lsbatch.h` as

```
struct submitReply {
    char   *queue;            Queue name the job was submitted to
    LS_LONG_INT badJobId;     dependCond contains badJobId but there is
                                  no such job
    char   *badJobName;       dependCond contains badJobName but
                                  there is no such job
    int    badReqIndx;        Index of a host or resource limit that caused
                                  an error
};
```

The last three variables in the structure submitReply are only used when the `lsb_submit()` or `lsb_modify()` fail.

For a complete description of the fields in the submitReply structure, see the `lsb_submit(3)` man page.

To submit a new job, fill out this data structure and then call `lsb_submit()`. The delOptions variable is ignored by LSF batch for `lsb_submit()`.

Example

The example job submission program below takes the job command line as an argument and submits the job to LSF batch. For simplicity, it is assumed that the job command does not have arguments.

```
/******************************************************
* LSBLIB -- Examples
*
* simple bsub
* This program submits a batch job to LSF
* It is the equivalent of using the "bsub" command without
* any options.
******************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <lsf/lsbatch.h>
#include "combine_arg.h"
    /* To use the function "combine_arg" to combine arguments on the
command line include its header file "combine_arg.h". */
int main(int argc, char **argv)
{
    struct submit req;           /* job specifications */
    memset(&req, 0, sizeof(req)); /* initializes req */
    struct submitReply  reply;  /* results of job submission */
    int  jobId;                 /* job ID of submitted job */
    int  i;
    /* initialize LSBLIB  and  get  the  configuration
    environment */
    if (lsb_init(argv[0]) < 0) {
        lsb_perror("simbsub: lsb_init() failed");
        exit(-1);
    }
    /* check if input is in the right format: "./simbsub
    COMMAND ARGUMENTS" */
    if (argc < 2) {
    fprintf(stderr, "Usage: simbsub command\n");
    exit(-1);
    }
    /* options and options2 are bitwise inclusive OR of some of
    the SUB_* flags */
    req.options = 0;
    req.options2 = 0;
    for (i = 0; i < LSF_RLIM_NLIMITS; i++)    /* resource
                                              limits are
                                              initialized to
                                              default */
        req.rLimits[i] = DEFAULT_RLIMIT;
    req.beginTime = 0;
        /* specific date and time to dispatch the job */
    req.termTime  = 0;
        /* specifies job termination deadline */
    req.numProcessors = 1;
/* initial number of processors needed by a (parallel) job */
    req.maxNumProcessors = 1;
/* max num of processors required to run the (parallel) job */
    req.command = combine_arg(argc,argv);
        /* command line of job */
printf("----------------------------------------------\n");
    jobId = lsb_submit(&req, &reply);
        /* submit the job with specifications */
    if (jobId < 0)
        /* if job submission fails, lsb_submit returns -1 */
    switch (lsberrno) {
        /* and sets lsberrno to indicate the error */
        case LSBE_QUEUE_USE:
        case LSBE_QUEUE_CLOSED:
            lsb_perror(reply.queue);
            exit(-1);
        default:
            lsb_perror(NULL);
            exit(-1);
    }
    exit(0);
}
/* main */
```

The above program will produce output similar to the following:

```
Job <5602> is submitted to default queue <default>.
```

Sample program explanations

Options and options2

```
req.options = 0;
```

```
req.options2 = 0;
```

The options and options2 fields of the submit structure are the bitwise inclusive OR of some of the `SUB_*` flags defined in `lsbatch.h`. These flags serve two purposes.

Some flags indicate which of the optional fields of the submit structure are present. Those that are not present have default values.

Other flags indicate submission options. For a description of these flags, see `lsb_submit(3)`.

Since options indicate which of the optional fields are meaningful, the programmer does not need to initialize the fields that are not chosen by options. All parameters that are not optional must be initialized properly.

numProcessors and maxNumProcessors

```
req.numProcessors = 1;
```

```
/* initial number of processors needed by a (parallel) job */
    req.maxNumProcessors = 1;
/* max number of processors required to run the (parallel) job */
```

numProcessors and maxNumProcessors are initialized to ensure only one processor is requested. They are defined in order to synchronize the job specification in `lsb_submit()` to the default used by `bsub`.

If the resReq field of the submit structure is `NULL`, then LSBLIB will try to obtain resource requirements for a command from the remote task list. If the task does not appear in the remote task list, then `NULL` is passed to LSF batch. `mbatchd` uses the default resource requirements with option `DFT_FROMTYPE` bit set when making a LSLIB call for host selection from LIM.

rLimits[LSF_RLIM_NLIMITS] and hostSpec

```
for (i = 0; i < LSF_RLIM_NLIMITS; i++)
```

```
        /* resource limits are initialized to default */
        req.rLimits[i] = DEFAULT_RLIMIT;
```

The default resource limit (`DEFAULT_RLIMIT`) defined in `lsf.h` are for no resource limits.

The constants used to index the rlimits array of the submit structure is defined in `lsf.h`. The resource limits currently supported by LSF batch are listed below.

Resource Limit	Index in `rlimits` Array
CPU time limit (in seconds)	`LSF_RLIMIT_CPU`
File size limit (in kilobytes)	`LSF_RLIMIT_FSIZE`
Data size limit (in kilobytes)	`LSF_RLIMIT_DATA`

Resource Limit	Index in `rlimits` Array
Stack size limit	`LSF_RLIMIT_STACK`
Core file size limit (in kilobytes)	`LSF_RLIMIT_CORE`
Resident memory size limit (in kilobytes)	`LSF_RLIMIT_RSS`
Number of open files limit	`LSF_RLIMIT_NOFILE`
Number of open files limit (for HP-UX)	`LSF_RLIMIT_OPEN_MAX`
Virtual memory limit (same as max swap memory)	`LSF_RLIMIT_SWAP`
Wall-clock time run limit	`LSF_RLIMIT_RUN`
Maximum num of processes a job can fork	`LSF_RLIMIT_PROCESS`
Thread number limit	`LSF_RLIMIT_THREAD`

The hostSpec field of the submit structure specifies the host model to use for scaling rlimits[LSF_RLIMIT_CPU] and rlimits[LSF_RLIMIT_RUN] (See `lsb_queueinfo (3)`). If hostSpec is `NULL`, the local host's model is assumed.

beginTime and termTime

```
req.beginTime = 0;/* specific date and time to dispatch the job */
```

```
    req.termTime  = 0;/* specifies job termination deadline */
```

If the beginTime field of the submit structure is 0, start the job as soon as possible.

A USR2 signal is sent if the job is running at termTime. If the job does not terminate within 10 minutes after being sent this signal, it is killed. If the termTime field of the submit structure is 0, the job is allowed to run until it reaches a resource limit.

lsberrno

The example below checks the value of `lsberrno` when `lsb_submit()` fails:

```
    if (jobId < 0)
        /* if job submission fails, lsb_submit returns -1 */
    switch (lsberrno) {
        /* and sets lsberrno to indicate the error */
    case LSBE_QUEUE_USE:
    case LSBE_QUEUE_CLOSED:
    lsb_perror(reply.queue);
    exit(-1);
    default:
    lsb_perror(NULL);
    exit(-1);
}
```

Different actions are taken depending on the type of the error. All possible error numbers are defined in `lsbatch.h`. For example, error number `LSBE_QUEUE_USE` indicates that the user is not authorized to use the queue. The error number `LSBE_QUEUE_CLOSED` indicates that the queue is closed.

Since a queue name was not specified for the job, the job is submitted to the default queue. The queue field of the submitReply structure contains the name of the queue to which the job was submitted.

The above program will produce output similar to the following:

```
Job <5602> is submitted to default queue <default>.
```

The output from the job is mailed to the user because the program did not specify a file name for the outFile parameter in the submit structure.

The program assumes that uniform user names and user ID spaces exist among all the hosts in the cluster. That is, a job submitted by a given user will run under the same user's account on the execution host. For situations where non-uniform user names and user ID spaces exist, account mapping must be used to determine the account used to run a job.

If you are familiar with the `bsub` command, it may help to know how the fields in the submit structure relate to the `bsub` command options. This is provided in the following table.

bsub Option	**submit Field**	**options**
`-J job_name_spec`	`jobName`	`SUB_JOB_NAME`
`-q queue_name`	`queue`	`SUB_QUEUE`
`-m host_name[+[pref_level]]`	`askedHosts`	`SUB_HOST`

`-n min_proc[,max_proc]`	`numProcessors,`
	`maxNumProcessors`	

`-R res_req`	`resReq`	`SUB_RES_REQ`
`-c cpu_limit[/host_spec]`	`rlimits[LSF_RLIMIT_CPU] / hostSpec **`	`SUB_HOST_SPEC (if host_spec is specified)`
`-W run_limit[/host_spec]`	`rlimits[LSF_RLIMIT_RUN] / hostSpec**`	`SUB_HOST_SPEC (if host_spec is specified)`
`-F file_limit`	`rlimits[LSF_RLIMIT_FSIZE]**`	
`-M mem_limit`	`rlimits[LSF_RLIMIT_RSS]**`	
`-D data_limit`	`rlimits[LSF_RLIMIT_DATA]**`	
`-S stack_limit`	`rlimits[LSF_RLIMIT_STACK**`	
`-C core_limit`	`rlimits[LSF_RLIMIT_CORE]**`	

`-k "chkpnt_dir [chkpnt_period]"`	`chkpntDir, chkpntPeriod`	`SUB_CHKPNT_DIR,`
		`SUB_CHKPNT_DIR (if chkpntPeriod is specified)`

bsub Option	submit Field	options
`-w depend_cond`	`dependCond`	`SUB_DEPEND_COND`
`-b begin_time`	`beginTime`	
`-t term_time`	`TermTime`	
`-i in_file`	`inFile`	`SUB_IN_FILE`
`-o out_file`	`outFile`	`SUB_OUT_FILE`
`-e err_file`	`errFile`	`SUB_ERR_FILE`
`-u mail_user`	`mailUser`	`SUB_MAIL_USER`
`-f "lfile op [rfile]"`	`xf`	
`-E "pre_exec_cmd [arg]"`	`preExecCmd`	`SUB_PRE_EXEC`
`-L login_shell`	`loginShell`	`SUB_LOGIN_SHELL`
`-P project_name`	`projectName`	`SUB_PROJECT_NAME`
`-G user_group`	`userGroup`	`SUB_USER_GROUP`
`-H`		`SUB2_HOLD*`
`-x`		`SUB_EXCLUSIVE`
`-r`		`SUB_RERUNNABLE`
`-N`		`SUB_NOTIFY_END`
`-B`		`SUB_NOTIFY_BEGIN`
`-I`		`SUB_INTERACTIVE`
`-Ip`		`SUB_PTY`
`-Is`		`SUB_PTY_SHELL`
`-K`		`SUB2_BSUB_BLOCK*`
`- X "except_cond::action"`	`exceptList`	`SUB_EXCEPT`
`-T time_event`	`timeEvent`	`SUB_TIME_EVENT`

* indicates a bitwise OR mask for options2.

** indicates -1 means undefined

Even if all the options are not used, all optional string fields must be initialized to the empty string. For a complete description of the fields in the submit structure, see the `lsb_submit (3)` man page.

To modify an already submitted job, fill out a new submit structure to override existing parameters, and use delOptions to remove option bits that were previously specified for the job. Modifying a submitted job is like re-submitting the job. Thus a similar program can be used to modify an existing job with minor changes. One additional parameter that must be specified for job modification is the job Id. The parameter delOptions can also be set if you want to clear some option bits that were previously set.

All applications that call `lsb_submit()` and `lsb_modify()` are subject to authentication constraints described in .

Batch job information

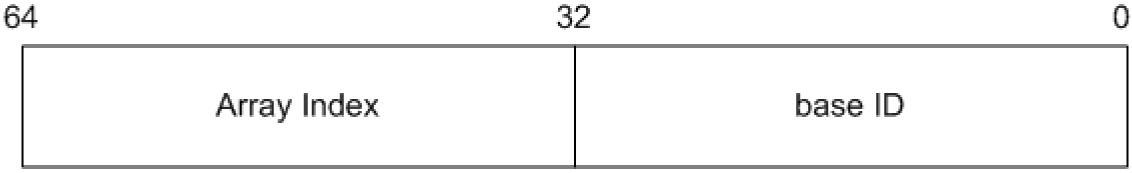
LSBLIB provides functions to get status information about batch jobs. Since there could be many thousands of jobs in the LSF batch system, getting all of this information in one message could use a lot of memory space. LSBLIB allows the application to open a stream connection and then read the job records one by one. This insures the memory space needed is always the size of one job record.

LSF batch job ID

LSF APIs supports 64-bit batch job ID. The LSF batch job ID will store in a 64-bit integer. It consists of two parts:

- Base ID
- Array index

The base ID is stored in the lower 32 bits. The array index is shared in the top 32 bits. The top 32 bits are only used when the underlying job is an array job.



LSBLIB provides the following C macros (defined in `lsbatch.h`) for manipulating job IDs:

```
LSB_JOBID(base_ID, array_index)   Yield an LSF batch job ID
LSB_ARRAY_IDX(job_ID)             Yield array index part of the job ID
LSB_ARRAY_JOBID(job_ID)           Yield the base ID part of the job ID
```

The function calls used to get job information are:

- `int lsb_openjobinfo(`*job_ID*`, `*jobName*`, `*user*`, `*queue*`, `*host*`, `*options*`);`
- `struct jobInfoEnt *lsb_readjobinfo(`*more*`);`
- `void lsb_closejobinfo(`*void*`);`

These functions are used to open a job information connection with `mbatchd`, read job records, and then close the job information connection.

lsb_openjobinfo()

`lsb_openjobinfo()` takes the following arguments:

```
LS_LONG_INT  jobId;            Select job with the given job Id
char  *jobName;                Select job(s) with the given job name
char  *user;                   Select job(s) submitted by the named user
                                   or user group
char  *queue;                  Select job(s) submitted to the named queue
char  *host;                   Select job(s) that are dispatched to the
                                   named host
int   options;                 Selection flags constructed from the bits
                                   defined in lsbatch.h
```

options parameter

The options parameter contains additional job selection flags defined in `lsbatch.h`. These are:

Flag Name	Flag Description
`ALL_JOB`	Select jobs matching any status, including unfinished jobs and recently finished jobs. LSF batch remembers finished jobs within the `CLEAN_PERIOD`, as defined in the `lsb.params` file.
`CUR_JOB`	Return jobs that have not finished yet
`DONE_JOB`	Return jobs that have finished recently.
`PEND_JOB`	Return jobs that are in the pending status.
`SUSP_JOB`	Return jobs that are in the suspended status.
`LAST_JOB`	Return jobs that are submitted most recently.
`JGRP_ARRAY_INFO`	Return job array information.

If options is 0, then the default is `CUR_JOB`.

`lsb_openjobinfo()` returns the total number of matching job records in the connection. On failure, it returns -1 and sets `lsberrno` to indicate the error.

lsb_readjobinfo()

`lsb_readjobinfo()` takes one argument:

```
int   *more;                 If not NULL, contains the remaining number of
                             jobs unread
```

Either this parameter or the return value from the `lsb_openjobinfo()` can be used to keep track of the number of job records that can be returned from the connection. This parameter is updated each time `lsb_readjobinfo()` is called.

jobInfoEnt structure

The jobInfoEnt structure returned by `lsb_readjobinfo()` is defined in `lsbatch.h` as:

```
struct jobInfoEnt {
    LS_LONG_INT  jobId;             job ID
    char         *user;             submission user
    int          status;            job status
    /* possible values for the status field */
#define JOB_STAT_PEND      0x01    job is pending
#define JOB_STAT_PSUSP     0x02    job is held
#define JOB_STAT_RUN       0x04    job is running
#define JOB_STAT_SSUSP     0x08    job is suspended by LSF batch system
#define JOB_STAT_USUSP     0x10    job is suspended by user
#define JOB_STAT_EXIT      0x20    job exited
#define JOB_STAT_DONE      0x40    job is completed successfully
#define JOB_STAT_PDONE     0x80    post job process done successfully
#define JOB_STAT_PERROR    0x100   post job process error
#define JOB_STAT_WAIT      0x200   chunk job waiting its execution turn
#define JOB_STAT_UNKWN     0x1000  unknown status
    int    *reasonTb;         pending or suspending reasons
    int    numReasons;        length of reasonTb vector
    int    reasons;           reserved for future use
    int    subreasons;        reserved for future use
    int    jobPid;            process Id of the job
    time_t submitTime;        time when the job is submitted
    time_t reserveTime;       time when job slots are reserved
    time_t startTime;         time when job is actually started
    time_t predictedStartTime;  job's predicted start time
    time_t endTime;           time when the job finishes
    time_t lastEvent;         last time event
    time_t nextEvent;         next time event
    int    duration;          duration time (minutes)
    float  cpuTime;           CPU time consumed by the job
    int    umask;             file mode creation mask for the job
    char   *cwd;              current working directory where job is
                                  submitted
    char   *subHomeDir;       submitting user’s home directory
    char   *fromHost;         host from which the job is submitted
    char   **exHosts;         host(s) on which the job executes
    int    numExHosts;        number of execution hosts
    float  cpuFactor;         CPU factor of the first execution host
    int    nIdx;              number of load indices in the loadSched and
                              loadStop vector
    float  *loadSched;        stop scheduling new jobs if this threshold is
                              exceeded
    float  *loadStop;         stop jobs if this threshold is exceeded
    struct submit submit;     job submission parameters
    int    exitStatus;        exit status
    int    execUid;           user ID under which the job is running
    char   *execHome;         home directory of the user denoted by
                                  execUid
    char   *execCwd;          current working directory where job is
                                  running
    char   *execUsername;     user name corresponds to execUid
    time_t jRusageUpdateTime; last time job's resource usage is updated
    struct jRusage runRusage; last updated job's resource usage
    int   jType;              job type
    /* Possible values for the jType field */
#define    JGRP_NODE_JOB        1  this structure stores a normal batch job
#define    JGRP_NODE_GROUP      2  this structure stores a job group
#define    JGRP_NODE_ARRAY      3  this structure stores a job array
    char  *parentGroup;       for job group use
    char  *jName;              if jType is JGRP_NODE_GROUP, then it is
                              job group name. Otherwise, it is the job's
                              name
    int   counter[NUM_JGRP_COUNTERS];
    /* index into the counter array, only used for job array */
#define     JGRP_COUNT_NJOBS   0   total jobs in the array
#define     JGRP_COUNT_PEND    1   number of pending jobs in the array
#define     JGRP_COUNT_NPSUSP  2   number of held jobs in the array
#define     JGRP_COUNT_NRUN    3   number of running jobs in the array
#define     JGRP_COUNT_NSSUSP  4   number of jobs suspended by the
                                   system in the array
#define     JGRP_COUNT_NUSUSP  5   number of jobs suspended by the
                                   user in the array
#define     JGRP_COUNT_NEXIT   6   number of exited jobs in the array
#define     JGRP_COUNT_NDONE   7   number of successfully completed jobs
    int     counter[NUM_JGRP_COUNTERS];
    u_short port;             service port of the job
    int    jobPriority;       job dynamic priority
    int    numExternalMsg;    number of external messages in the job
    struct jobExternalMsgReply **externalMsg;
    int     clusterId;
    char   *detailReason;     Detail reason field
    float   idleFactor;
    int     exceptMask;       Job exception mask
    char   *additionalInfo;   Arbitrary job information string
                             currently used by rms_rid and rms_alloc
    int     exitInfo;         Termination reason
    int    warningTimePeriod;  Warning time in seconds, -1 if
                               unspecified
    char   *warningAction;     Warning action, SIGNAL | CHKPNT |
                               command, NULL if unspecified
    char   *chargedSAAP;       SAAP charged for job
    char   *execRusage;        The rusage satisfied at job runtime
    time_t rsvInActive;          Time when AR was expired or deleted
    int    numLicense;            Number of licenses reported from LS
    char   **licenseNames;     LS license names
    float  aps;                Absolute priority value
    float  adminAps;           Static aps value set by admin
    int    runTime;            Job's real runtime
    int reserveCnt;  Number of resource types reserved by this job
    struct reserveItem *items;  Detail reservation information for
                                each kind of resource
    float  adminFactorVal;      Admin factor value
    int    resizeMin;   Pending resize min. 0, if no resize pending
    int    resizeMax;   Pending resize max. 0, if no resize pending
    time_t resizeReqTime;       Time when pending request was issued
    int    jStartNumExHosts;    Number of hosts when job starts
    char   **jStartExHosts;     Host list when job starts
   time_t lastResizeTime;       Last time job allocation changed
};
```

jobInfoEnt can store a job array as well as a non-array batch job, depending on the value of jType field, which can be either `JGRP_NODE_JOB` or `JGRP_NODE_ARRAY`.

lsb_closejobinfo()

Call `lsb_closejobinfo()`after receiving all job records in the connection.

Example

Below is an example of a simplified `bjobs` command. This program displays all pending jobs belonging to all users.

```
/******************************************************

* LSBLIB -- Examples

*

* simple bjobs

* Submit command as an lsbatch job with no options set

* and retrieve the job info

* It is similar to the "bjobs" command with no options.

******************************************************/
```

```
#include <stdio.h>
#include <lsf/lsbatch.h>
#include "submit_cmd.h"
int main(int argc, char **argv)
{
    /* variables for simulating submission */
    struct submit req;            /* job specifications */
    memset(&req, 0, sizeof(req)); /* initializes req */
    struct submitReply  reply; /* results of job submission */
    int  jobId;                /* job ID of submitted job */
    /* variables for simulating bjobs command */
    int  options = PEND_JOB;    /* the status of the jobs
                                whose info is returned */
    char *user="all";         /* match jobs for all users */
    struct jobInfoEnt *job;     /* detailed job info */
    int more;                   /* number of remaining jobs
                                unread */
    /* initialize LSBLIB  and  get  the  configuration
    environment */
    if (lsb_init(argv[0]) < 0) {
        lsb_perror("simbjobs: lsb_init() failed");
        exit(-1);
    }
    /* check if input is in the right format:
     * "./simbjobs COMMAND ARGUMENTS" */
    if (argc < 2) {
        fprintf(stderr, "Usage: simbjobs command\n");
        exit(-1);
    }
    jobId = submit_cmd(&req, &reply, argc, argv);
        /* submit a job */
    if (jobId < 0)                      /* if job submission
                                        fails, lsb_submit
                                        returns -1 */
        switch (lsberrno) {
        /* and sets lsberrno to indicate the error */
        case LSBE_QUEUE_USE:
        case LSBE_QUEUE_CLOSED:
            lsb_perror(reply.queue);
            exit(-1);
        default:
            lsb_perror(NULL);
            exit(-1);
}
    /* gets the total number of pending job. Exits if failure */
    if (lsb_openjobinfo(0, NULL, user, NULL, NULL, options)<0) {
        lsb_perror("lsb_openjobinfo");
        exit(-1);
    }
    /* display all pending jobs */
    printf("All pending jobs submitted by all users:\n");
    for (;;) {
        job = lsb_readjobinfo(&more);   /* get the job details */
        if (job == NULL) {
        lsb_perror("lsb_readjobinfo");
        exit(-1);
    }
        printf("%s",ctime(&job->submitTime));
        /* submission time of job */
        printf("Job <%s> ", lsb_jobid2str(job->jobId));
        /* job ID */
        printf("of user <%s>, ", job->user);
        /* user that submits the job */
        printf("submitted from host <%s>\n", job->fromHost);            /*
name of sumbission host */
        /* continue to display if there is remaining job */
        if (!more)
        /* if there are no remaining jobs undisplayed,
           exits */
        break;
    }
    /* when finished to display the job info, close the
    connection to the mbatchd */
    lsb_closejobinfo();
    exit(0);
}
```

The above program will produce output similar to the following:

```
All pending jobs submitted by all users:
Mon Mar 1 10:34:04 EST 1996
Job <123> of user <john>, submitted from host <orange>
Mon Mar 1 11:12:11 EST 1996
Job <126> of user <john>, submitted from host <orange>
Mon Mar 1 14:11:34 EST 1996
Job <163> of user <ken>, submitted from host <apple>
Mon Mar 1 15:00:56 EST 1996
Job <199> of user <tim>, submitted from host <pear>
```

Use `lsb_pendreason()`, to print out the reasons why the job is still pending See `lsb_pendreason(3)` for details.

Job manipulation

Users manipulate jobs in different ways, after a job has been submitted. It can be suspended, resumed, killed, or sent arbitrary signal jobs.

All applications that manipulate jobs are subject to authentication provisions.

Send a signal to a job

Users can send signals to submitted jobs. If the job has not been started, you can send `KILL`, `TERM`, `INT`, and `STOP` signals. These signals cause the job to be cancelled (`KILL`, `TERM`, `INT`) or suspended (`STOP`). If the job has already started, then any signal can be sent to the job.

lsb_signaljob()

`lsb_signaljob()` sends a signal to a job:

```
int lsb_signaljob(jobId, sigValue);
LS_LONG_INT  jobId;          Select job with the given job Id
int sigValue;                Signal sent to the job
```

The jobId and sigValue parameters are self-explanatory.

Example

The following example takes a job ID as the argument and sends a `SIGSTOP` signal to the job.

```
/******************************************************

* LSBLIB -- Examples

*

* simple bstop

* The program takes a job ID as the argument and sends a * SIGSTOP signal to the job

******************************************************/
```

```
#include <stdio.h>
#include <lsf/lsbatch.h>
#include <stdlib.h>
#include <signal.h>
int main(int argc, char **argv)
{
    /* check if input is in the right format: "simbstop JOBID" */
    if (argc != 2) {
        printf("Usage: %s jobId\n", argv[0]);
        exit(-1);
    }
    /* initialize LSBLIB and get the configuration environment */
    if (lsb_init(argv[0]) < 0) {
        lsb_perror("lsb_init");
        exit(-1);
    }
    /* send the SIGSTOP signal and check if lsb_signaljob()
    runs successfully */
    if (lsb_signaljob(atoi(argv[1]), SIGSTOP) <0) {
        lsb_perror("lsb_signaljob");
        exit(-1);
    }
    printf("Job %s is signaled\n", argv[1]);
    exit(0);
    }
```

On success, the function returns 0. On failure, it returns -1 and sets `lsberrno` to indicate the error.

Switch a job to a different queue

A job can be switched to a different queue after submission. This can be done even after the job has already started.

lsb_switchjob()

Use `lsb_switchjob()` to switch a job from one queue to another:

```
int lsb_switchjob(jobId, queue);
LS_LONG_INT jobId;            Select job with the given job Id
char *queue                   Name of the queue for the new job
```

Example

Below is an example program that switches a specified job to a new queue.

```
/*******************************************************
* LSBLIB -- Examples
*
* simple bstop
* The program switches a specified job to a new queue.
*******************************************************/
#include <stdio.h>
#include <lsf/lsbatch.h>
#include <stdlib.h>
int main(int argc, char **argv)
{
    /* check if the input is in the right format: "./simbstop
    JOBID QUEUENAME" */
    if (argc != 3) {
        printf("Usage: %s jobId new_queue\n", argv[1]);
        exit(-1);
    }
    /* initialize LSBLIB and get the configuration environment */
    if (lsb_init(argv[0]) <0) {
        lsb_perror("lsb_init");
        exit(-1);
    }
    /* switch the job to the new queue and check for success */
    if (lsb_switchjob(atoi(argv[1]), argv[2]) < 0) {
        lsb_perror("lsb_switchjob");
        exit(-1);
    }
    printf("Job %s is switched to new queue <%s>\n", argv[1],            argv
[2]);
    exit(0);
}
```

On success, `lsb_switchjob()` returns 0. On failure, it returns -1 and sets `lsberrno` to indicate the error.

Force a job to run

After a job is submitted to the LSF batch system, it remains pending until LSF batch runs it (for details on the factors that govern when and where a job starts to run, see Administering Platform LSF).

lsb_runjob()

A job can be forced to run on a specified list of hosts immediately using the following LSBLIB function:

```
int lsb_runjob (struct runJobRequest *runReq)
```

runJobReq Structure

`lsb_runjob()` takes the runJobRequest structure, which is defined in `lsbatch.h`:

```
struct runJobRequest {
    LS_LONG_INT  jobId;              Job ID of the job to start
    int          numHosts;           Number of hosts to run the job on
    char         **hostname;         Host names where jobs run
#define RUNJOB_OPT_NORMAL     0x01
#define RUNJOB_OPT_NOSTOP     0x02
#define RUNJOB_OPT_PENDONLY   0x04     Pending jobs only, no finished jobs
#define RUNJOB_OPT_FROM_BEGIN 0x08     Checkpoint jobs only, from beginning
#define RUNJOB_OPT_FREE       0x10     brun to use free CPUs only
    int          options;            Run job request options
    int          *slots;             Number of slots per host
}
```

To force a job to run, the job must have been submitted and in either PEND or FINISHED state. Only the LSF administrator or the owner of the job can start the job. `lsb_runjob()` restarts a job in FINISHED status.

A job can be run without any scheduling constraints such as job slot limits. If the job is started with the options field being 0 or RUNJOB_OPT_NORMAL, then the job is subject to the:

- Run windows in the default queue
- Queue threshold
- Execution hosts for the job

To override a started, use RUNJOB_OPT_NOSTOP and the job will not be stopped due to the above mentioned load conditions. However, all LSBLIB's job manipulation APIs can still be applied to the job.

Example

The following is an example program that runs a specified job on a host that has no batch job running.

```
/******************************************************

* LSBLIB -- Examples

*

* simple brun

* The program takes a job ID as the argument and runs that

* job on a vacant hosts

******************************************************/
```

```
#include <stdio.h>
#include <lsf/lsbatch.h>
#include <stdlib.h>
int main(int argc, char **argv)
{
    struct hostInfoEnt  *hInfo;  /* host information */
    int numHosts = 0;            /* number of hosts */
    int i;
    struct runJobRequest runJobReq;
        /* specification for the job to be run */
    /* check if the input is in the right format: "./simbrun
    JOBID" */
    if (argc != 2) {
        printf("Usage: %s jobId\n", argv[0]);
        exit(-1);
    }
    /* initialize LSBLIB and get the configuration environment */
    if (lsb_init(argv[0]) < 0) {
        lsb_perror("lsb_init");
        exit(-1);
    }
    /* get host information */
    hInfo = lsb_hostinfo(NULL, &numHosts);
    if (hInfo == NULL) {
        lsb_perror("lsb_hostinfo");
        exit(-1);
    }
    /* find a vacant host */
    for (i = 0; i < numHosts; i++) {
       if (hInfo[i].hStatus & (HOST_STAT_BUSY |
                              HOST_STAT_WIND |
                              HOST_STAT_DISABLED |
                              HOST_STAT_LOCKED |
                              HOST_STAT_FULL |
                              HOST_STAT_NO_LIM |
                              HOST_STAT_UNLICENSED |
                              HOST_STAT_UNAVAIL |
                              HOST_STAT_UNREACH))
            continue;
        /* found a vacant host */
        if (hInfo[i].numJobs == 0)
            break;
    }
    /* return error message when there is no vacant host found */
    if (i == numHosts) {
        fprintf(stderr, "Cannot find vacate host to run job
                < %s >\n", argv[1]);
        exit(-1);
    }
    /* define the specifications for the job to be run (The job
    can be stopped due to load conditions) */
    runJobReq.jobId = atoi(argv[1]);
    runJobReq.options = 0;
    runJobReq.numHosts = 1;
    runJobReq.hostname = (char **)malloc(sizeof(char*));
    runJobReq.hostname[0] = hInfo[i].host;
    /* run the job and check for the success */
    if (lsb_runjob(&runJobReq) < 0) {
        lsb_perror("lsb_runjob");
        exit(-1);
    }
    exit (0);
}
```

On success, `lsb_runjob()` returns 0. On failure, returns -1 and sets `lsberrno` to indicate the error.

LSF batch event files

LSF batch saves a lot of valuable information about the system and jobs. Such information is logged by `mbatchd` in the files `lsb.events` and `lsb.acct` under the directory `$LSB_SHAREDIR/your_cluster/logdir`, where `LSB_SHAREDIR` is defined in the `lsf.conf` file and `your_cluster` is the name of your Platform LSF cluster.

`mbatchd` logs such information for several purposes.

- Some of the events serve as the backup of `mbatchd`'s memory. In case `mbatchd` crashes, all critical information from the event file can then be used by the newly started `mbatchd` to restore the current state of LSF batch.
- The events can be used to produce historical information about the LSF batch system and user jobs.
- Such information can be used to produce accounting or statistic reports.

Caution:

The `lsb.events` file contains critical user job information. Never use your program to modify `lsb.events`. Writing into this file may cause the loss of user jobs.

lsb_geteventrec()

LSBLIB provides a function to read information from these files into a well-defined data structure:

```
struct eventRec *lsb_geteventrec(log_fp, lineNum)
FILE  *log_fp;              File handle for either an event log
                            file or job log file
int   *lineNum;             Line number of the next event
                              record
```

The parameter log_fp is returned by a successful `fopen()` call. The content in lineNum is modified to indicate the line number of the next event record in the log file on a successful return. This value can then be used to report the line number when an error occurs while reading the log file. This value should be initiated to 0 before `lsb_geteventrec()` is called for the first time.

eventRec Structure

`lsb_geteventrec()` returns the following data structure:

```
struct eventRec {
    char version[MAX_VERSION_LEN];    Version number of the mbatchd
    int type;                         Type of the event
    time_t eventTime;                    Event time stamp
    union eventLog eventLog;          Event data
};
```

The event type is used to determine the structure of the data in eventLog. LSBLIB remembers the storage allocated for the previously returned data structure and automatically frees it before returning the next event record.

`lsb_geteventrec()` returns `NULL` and sets `lsberrno` to `LSBE_EOF` when there are no more records in the event file.

Events are logged by `mbatchd` for different purposes. There are job-related events and system-related events. Applications can choose to process certain events and ignore other events. For example, the `bhist` command processes job-related events only. The currently available event types are listed below.

Event Type	Description
`EVENT_JOB_NEW`	Submit new job
`EVENT_JOB_START`	`mbatchd` is trying to start a job

Event Type	**Description**
`EVENT_JOB_STATUS`	Job status change event
`EVENT_JOB_SWITCH`	Job switched to another queue
`EVENT_JOB_MOVE`	Move a pending job’s position within a queue
`EVENT_QUEUE_CTRL`	Queue status changed by Platform LSF administrator (bqc operation)
`EVENT_HOST_CTRL`	Host status changed by Platform LSF administrator (bhc operation)
`EVENT_MBD_START`	New `mbatchd` start event
`EVENT_MBD_DIE`	Log parameters before `mbatchd` die
`EVENT_MBD_UNFULFILL`	`mbatchd` has an action to be fulfilled
`EVENT_JOB_FINISH`	Job has finished (logged in `lsb.acct` only)
`EVENT_LOAD_INDEX`	Complete list of load index names
`EVENT_MIG`	Job has migrated
`EVENT_PRE_EXEC_START`	The pre-execution command started
`EVENT_JOB_ROUTE`	The job has been routed to NQS
`EVENT_JOB_MODIFY`	The job’s parameters have been modified
`EVENT_JOB_SIGNAL`	Signal/delete a job
`EVENT_CAL_NEW`	Add new calendar to the system *
`EVENT_CAL_MODIFY`	Calendar modified *
`EVENT_CAL_DELETE`	Calendar deleted *
`EVENT_JOB_FORCE`	Forcing a job to start on specified hosts (`brun` operation)
`EVENT_JOB_FORWARD`	Job forwarded to another cluster
`EVENT_JOB_ACCEPT`	Job from a remote cluster dispatched
`EVENT_STATUS_ACK`	Job status successfully sent to submission cluster
`EVENT_JOB_EXECUTE`	Job started successfully on the execution host
`EVENT_JOB_MSG`	Send a message to a job
`EVENT_JOB_MSG_ACK`	The message has been delivered.
`EVENT_JOB_REQUEUE`	Job is requeued
`EVENT_JOB_OCCUPY_REQ`	Submission `mbatchd` logs this after sending an occupy request to execution `mbatchd`
`EVENT_JOB_VACATED`	Submission `mbatchd` logs this event after all execution `mbatchds` have vacated the occupied hosts for the job.

Event Type	Description
`EVENT_JOB_SIGACT`	An signal action on a job has been initiated or finished
`EVENT_JOB_START_ACCEPT`	Job accepted by `sbatchd`
`EVENT_SBD_JOB_STATUS`	`sbatchd`'s new job status
`EVENT_CAL_UNDELETE`	Undeleted a calendar in the system
`EVENT_JOB_CLEAN`	Job is cleaned out of the core
`EVENT_JOB_EXCEPTION`	Job exception was detected
`EVENT_JGRP_ADD`	Adding a new job group
`EVENT_JGRP_MOD`	Modifying a job group
`EVENT_JGRP_CNT`	Controlling a job group
`EVENT_LOG_SWITCH`	Switching the event file `lsb.events`
`EVENT_JOB_MODIFY2`	Job modification request
`EVENT_JGRP_STATUS`	Log job group status
`EVENT_JOB_ATTR_SET`	Job attributes have been set
`EVENT_JOB_EXT_MSG`	Send an external message to a job
`EVENT_JOB_ATTA_DATA`	Update data status of a message for a job
`EVENT_JOB_CHUNK`	Insert one job to a chunk
`EVENT_SBD_UNREPORTED_STATUS`	Save unreported `sbatchd` status
`EVENT_ADRSV_FINISH`	An advanced reservation expired.
`EVENT_HGHOST_CTRL`	Dynamic host group control changes.
`EVENT_CPUPROFILE_STATUS`	Save current CPU allocation on service partition.
`EVENT_DATA_LOGGING`	Write a data logging file.
`EVENT_JOB_RUN_RUSAGE`	Write job ruasage to `lsb.stream`.
`EVENT_END_OF_STREAM`	Close stream and open new stream.
`EVENT_SLA_RECOMPUTE`	Re-evaluate SLA goal.
`EVENT_METRIC_LOG`	Write performance metrics to `lsb.stream`.
`EVENT_TASK_FINISH`	Write a task finish log to `ssched.acct`.
`EVENT_JOB_RESIZE_NOTIFY_START`	Job resize allocation made.
`EVENT_JOB_RESIZE_NOTIFY_ACCEPT`	Job resize notification action initialized.

Event Type	Description
`EVENT_JOB_RESIZE_NOTIFY_DONE`	Job resize notification action completed.
`EVENT_JOB_RESIZE_RELEASE`	Job resize release request received.
`EVENT_JOB_RESIZE_CANCEL`	Job resize cancel request received.
`EVENT_JOB_RESIZE`	Job resize event for `lsb.acct`.

* Available only if the Platform JobScheduler component is enabled.

Tip:

The `lsb.acct` file uses only `EVENT_JOB_FINISH`. `lsb.events` file uses all other event types. For detailed formats of these log files, see `lsb.events(5)` and `lsb.acct(5)`.

eventLog Union

Each event type corresponds to a different data structure in the union:

```
union  eventLog {
    struct jobNewLog     jobNewLog;      EVENT_JOB_NEW
    struct jobStartLog   jobStartLog;    EVENT_JOB_START
    struct jobStatusLog  jobStatusLog;   EVENT_JOB_STATUS
    struct jobSwitchLog  jobSwitchLog;   EVENT_JOB_SWITCH
    struct jobMoveLog    jobMoveLog;     EVENT_JOB_MOVE
    struct queueCtrlLog  queueCtrlLog;   EVENT_QUEUE_CTRL
    struct hostCtrlLog   hostCtrlLog;    EVENT_HOST_CTRL
    struct mbdStartLog   mbdStartLog;    EVENT_MBD_START
    struct mbdDieLog     mbdDieLog;      EVENT_MBD_DIE
    struct unfulfillLog  unfulfillLog;   EVENT_MBD_UNFULFILL
    struct jobFinishLog  jobFinishLog;   EVENT_JOB_FINISH
    struct loadIndexLog  loadIndexLog;   EVENT_LOAD_INDEX
    struct migLog        migLog;         EVENT_MIG
    struct calendarLog   calendarLog;    Shared by all calendar events
    struct jobForceRequestLog jobForceRequestLog
                                               EVENT_JOB_FORCE
    struct jobForwardLog jobForwardLog;  EVENT_JOB_FORWARD
    struct jobAcceptLog  jobAcceptLog;   EVENT_JOB_ACCEPT
    struct statusAckLog  statusAckLog;   EVENT_STATUS_ACK
    struct signalLog     signalLog;      EVENT_JOB_SIGNAL
    struct jobExecuteLog jobExecuteLog;  EVENT_JOB_EXECUTE
    struct jobRequeueLog jobRequeueLog;  EVENT_JOB_REQUEUE
    struct sigactLog sigactLog;          EVENT_JOB_SIGACT
    struct jobStartAcceptLog jobStartAcceptLog
                                             EVENT_JOB_START_ACCEPT
    struct jobMsgLog     jobMsgLOg;      EVENT_JOB_MSG
    struct jobMsgAckLog  jobMsgAckLog;   EVENT_JOB_MSG_ACK
    struct chkpntLog     chkpntLog;      EVENT_CHKPNT
    struct jobOccupyReqLog jobOccupyReqLog;
                                               EVENT_JOB_OCCUPY_REQ
    struct jobVacatedLog jobVacatedLog;  EVENT_JOB_VACATED
    struct jobCleanLog   jobCleanLog;    EVENT_JOB_CLEAN
    struct jobExceptionLog jobExceptionLog;
                                               EVENT_JOB_EXCEPTION
    struct jgrpNewLog    jgrpNewLog;     EVENT_JGRP_ADD
    struct jgrpCtrlLog   jgrpCtrlLog;    EVENT_JGRP_CTR
    struct logSwitchLog  logSwitchLog;   EVENT_LOG_SWITCH
    struct jobModLog     jobModLog;      EVENT_JOB_MODIFY
    struct jgrpStatusLog jgrpStatusLog;  EVENT_JGRP_STATUS
    struct jobAttrSetLog jobAttrSetLog;  EVENT_JOB_ATTR_SET
    struct jobExternalMsgLog jobExternalMsgLog;
                                                EVENT_JOB_EXT_MSG
    struct jobChunkLog   jobChunkLog;    EVENT_JOB_CHUNK
    struct sbdUnreportedStatusLog sbdUnreportedStatusLog;
                                      EVENT_SBD_UNREPORTED_STATUS
    struct rsvFinishLog rsvFinishLog;
    struct hgCtrlLog hgCtrlLog;
    struct cpuProfileLog cpuProfileLog;
    struct dataLoggingLog dataLoggingLog;
    struct jobRunRusageLog   jobRunRusageLog;
    struct eventEOSLog       eventEOSLog;
    struct slaLog            slaLog;
    struct perfmonLog     perfmonLog;
    struct taskFinishLog     taskFinishLog;
    struct jobResizeNotifyStartLog    jobResizeNotifyStartLog;
    struct jobResizeNotifyAcceptLog   jobResizeNotifyAcceptLog;
    struct jobResizeNotifyDoneLog jobResizeNotifyDoneLog;
    struct jobResizeReleaseLog jobResizeReleaseLog;
    struct jobResizeCancelLog jobResizeCancelLog;
    struct jobResizeLog  jobResizeLog;
};
```

The detailed data structures in the above union are defined in `lsbatch.h` and described in `lsb_geteventrec(3)`.

Example

Below is an example program that takes an argument as job name and displays a chronological history about all jobs matching the job name. This program assumes that the `lsb.events` file is in `/local/lsf/work/cluster1/logdir`.

```
/******************************************************

* LSBLIB -- Examples

*

* get event record

* The program takes a job name as the argument and returns

* the information of the job with this given name

******************************************************/
```

```
#include <stdio.h>
#include <string.h>
#include <time.h>
#include <lsf/lsbatch.h>
int main(int argc, char **argv)
{
    char *eventFile =             "/local/lsf/mnt/work/cluster1/logdir/lsb.events";
       /*location of lsb.events*/
    FILE *fp;/* file handler for lsb.events */
    struct eventRec *record;
  /* pointer to the return struct of lsb_geteventrec() */
    int  lineNum = 0;/* line number of next event */
    char *jobName = argv[1];/* specified job name */
    int  i;
    struct jobNewLog *newJob;/* new job event record */
    struct jobStartLog *startJob;/* start job event record */
    struct jobStatusLog *statusJob;
        /* job status change event record */
    /* check if the input is in the right format:     "./geteventrec JOBNAME" */
    if (argc != 2) {
        printf("Usage: %s job name\n", argv[0]);
        exit(-1);
    }
    /* initialize LSBLIB and get the configuration environment */
    if (lsb_init(argv[0]) < 0) {
        lsb_perror("lsb_init");
        exit(-1);
    }
    /* open the file for read */
    fp = fopen(eventFile, "r");
    if (fp == NULL) {
        perror(eventFile);
        exit(-1);
    }
    /* get events and print out the information of the event
    records with the given job name in different format */
    for (;;) {
        record = lsb_geteventrec(fp, &lineNum);
        if (record == NULL) {
            if (lsberrno == LSBE_EOF)
                exit(0);
            lsb_perror("lsb_geteventrec");
            exit(-1);
        }
        /* find the record with the given job name */
        if (record->eventLog.jobNewLog.jobName==NULL)
            continue;
        if (strcmp(record->eventLog.jobNewLog.jobName, jobName) != 0)
            continue;
        else
            switch (record->type) {
        case EVENT_JOB_NEW:
             newJob = &(record->eventLog.jobNewLog);
                printf("%sJob <%d> submitted by <%s> from <%s>                       to <%s> queue
\n", ctime(&record->                      eventTime), newJob->jobId, newJob->
userName, newJob->fromHost, newJob->                      queue);
            continue;
        case EVENT_JOB_START:
            startJob = &(record->eventLog.jobStartLog);
                printf("%sJob <%d> started on ", ctime(&record->                      eventTime),
newJob->jobId);
            for (i=0; i<startJob->numExHosts; i++)
                printf("<%s> ", startJob->execHosts[i]);
            printf("\n");
            continue;
        case EVENT_JOB_STATUS:
            statusJob = &(record->eventLog.jobStatusLog);
                printf("%sJob <%d> status changed to: ",                       ctime(&record-
>eventTime), statusJob->                       jobId);
                switch(statusJob->jStatus) {
        case JOB_STAT_PEND:
                printf("pending\n");
                continue;
                case JOB_STAT_RUN:
                printf("running\n");
                continue;
        case JOB_STAT_SSUSP:
        case JOB_STAT_USUSP:
        case JOB_STAT_PSUSP:
                printf("suspended\n");
                continue;
        case JOB_STAT_UNKWN:
                printf("unknown (sbatchd unreachable)\n");
                continue;
        case JOB_STAT_EXIT:
                printf("exited\n");
                continue;
        case JOB_STAT_DONE:
                printf("done\n");
                continue;
        default:
               printf("\nError: unknown job status %d\n",                     statusJob->jStatus);
                continue;
        }
        default:
        /* Only display a few selected event types */
            continue;
            }
    }
    exit(0);
}
```

Tip:

In the above program, events that are of no interest are skipped. The job status codes are defined in `lsbatch.h`. The `lsb.acct` file stores job accounting information, which allows `lsb.acct` to be processed similarly. Since currently there is only one event type (EVENT_JOB_FINISH) in `lsb.acct`, processing is simpler than in the above example.

CHAPTER

4

Advanced Programming Topics

Load information for selected load indices

To get load information from the LIM: Depending on the size of your LSF cluster and the frequency at which the `ls_load()` function is called, returning load information of all the hosts can produce unnecessary overhead.

LSLIB provides `ls_loadinfo()` call that allows an application to specify a selected number of load indices and get only those load indices that are of interest to the application.

List all load index names

Since LSF allows a site to install an ELIM to collect additional load indices, the names and the total number of load indices are often dynamic and have to be found out at run time unless the application is only using the built-in load indices.

Example

Below is an example routine that returns a list of all available load index names and the total number of load indices.

```
#include <lsf/lsf.h>
char **getIndexList(int *listsize)
{
    struct lsInfo *lsInfo = (struct lsInfo *) malloc (sizeof
    (struct lsInfo));
    static char *nameList[268];
    static int first = 1;
    int i;
    if (first) {
        /* only need to do so when called for the first time */
        lsInfo = ls_info();
        if (lsInfo == NULL)
            return (NULL);
        first = 0;
    }
    if (listsize != NULL)
        *listsize = lsInfo->numIndx;
    for (i=0; i<lsInfo->numIndx; i++)
        nameList[i] = lsInfo->resTable[i].name;
    return (nameList);
}
```

The above code fragment returns a list of load index names currently installed in the LSF cluster. The content of listSize will be modified to the total number of load indices. If `ls_info()` fails, then the program returns `NULL`. The data structure returned by `ls_info()`contains all the load index names before any other resource names. The load index names start with the 11 built-in load indices followed by site external load indices (through ELIM).

Display selected load indices

By providing a list of load index names to an LSLIB function, you can get the load information about the specified load indices.

ls_loadinfo()

The following example shows how you can display the values of the external load indices. This program uses `ls_loadinfo()`:

```
struct hostLoad *ls_loadinfo(resreq, numhosts,
options,                             fromhost, hostlist, listsize,
                            namelist)
```

The parameters for this routine are:

```
char *resreq;                 Resource requirement
int *numhosts;                Return parameter, number of hosts returned
int options;                  Host and load selection options
char *fromhost;               Used only if DFT_FROMTYPE is set in options
char **hostlist;              A list of candidate hosts for selection
int listsize;                 Number of hosts in hostlist
char ***namelist;             Input/output parameter -- load index name list
```

`ls_loadinfo()` is similar to `ls_load()` except that `ls_loadinfo()` allows an application to supply both a list of load indices and a list of candidate hosts. If both of namelist and hostlist are `NULL`, then it operates in the same way as `ls_load()` function.

The parameter namelist allows an application to specify a list of load indices of interest. The function then returns only the specified load indices. On return, this parameter is modified to point to another name list that contains the same set of load index names. This load index is in a different order to reflect the mapping of index names and the actual load values returned in the hostLoad array:

Example

```
#include <stdio.h>
#include <lsf/lsf.h>
/*include the header file with the getIndexList function here*/
main()
{
    struct hostLoad *load;
    char **loadNames;
    int numIndx;
    int numUsrIndx;
    int nHosts;
    int i;
    int j;
    loadNames = getIndexList(&numIndx);
    if (loadNames == NULL) {
        ls_perror("Unable to get load index names\n");
        exit(-1);
    }
    numUsrIndx = numIndx - 11;  /* this is the total num of
                                site defined indices*/
    if (numUsrIndx == 0) {
        printf("No external load indices defined\n");
        exit(-1);
    }
    loadNames += 11;  /* skip the 11 built-in load index names */

    load = ls_loadinfo(NULL, &nHosts, 0, NULL, NULL, 0,
&loadNames);
    if (load == NULL) {
        ls_perror("ls_loadinfo");
        exit(-1);
    }
    printf("Report on external load indices\n");
    for (i=0; i<nHosts; i++) {
        printf("Host %s:\n", load[i].hostName);
        for (j=0; j<numUsrIndx; j++)
            printf("index name: %s, value %5.0f\n",
                   loadNames[j], load[i].li[j]);
    }
}
```

The above program uses the `getIndexList()` function described in the previous example program to get a list of all available load index names. Sample output from the above program follows:

```
Report on external load indices
Host hostA:
        index name: usr_tmp, value 87
        index name: num_licenses, value 1
Host hostD:
        index name: usr_tmp, value 18
        index name: num_licenses, value 2
```

Parallel applications

LSF provides job placement and remote execution support for parallel applications. A master LIM's host selection or placement service can return an array of good hosts for an application. The application can then use remote execution service provided by RES to run tasks on these hosts concurrently.

This section contains samples of how to write a parallel application using LSLIB.

ls_rtask() function

You can use of `ls_rexecv()` for remote execution. You can also use `ls_rtask()` for remote execution. `ls_rtask()`and `ls_rexecv()` differ in how the server host behaves.

`ls_rexecv()` is useful when the server host does not need to do anything but wait for the remote task to finish. After initiating the remote task, `ls_rexecv()` replaces the current program with the Network I/O Server (NIOS) by calling `execv()`. The NIOS then handles the rest of the work on the server host: delivering input/output between local terminal and remote task and exiting with the same status as the remote task. `ls_rexecv()` is considered to be the remote execution version of the UNIX `execv()` system call.

ls_rtask()

`ls_rtask()` provides more flexibility if the server host has to do other things after the remote task is initiated. For example, the application may want to start more than one task on several hosts. Unlike `ls_rexecv()`, `ls_rtask()` returns immediately after the remote task is started. The syntax of `ls_rtask()` is:

```
int ls_rtask(host, argv, options)
```

The parameters are:

```
char *host;                 Name of the remote host to start task on
char **argv;                Program name and arguments
int  options;                Remote execution options
```

options parameter

The options parameter is similar to that of the `ls_rexecv()` function. `ls_rtask()` returns the task ID of the remote task which is used by the application to differentiate multiple outstanding remote tasks. When a remote task finishes, the status of the remote task is sent back to the NIOS running on the local host, which then notifies the application by issuing a `SIGUSR1` signal. The application can then call `ls_rwait()` to collect the status of the remote task. The `ls_rwait()` behaves in much the same way as the `wait(2)` system call. Consider `ls_rtask()` as a combination of remote `fork()` and `execv()`.

Tip:

Applications calling `ls_rtask()` must set up a signal handler for the `SIGUSR1` signal, or the application could be killed by `SIGUSR1`.

You need to be careful if your application handles `SIGTSTP`, `SIGTTIN`, or `SIGTTOU`. If handlers for these signals are `SIG_DFL`, the `ls_rtask()` function automatically installs a handler for them to properly coordinate with the NIOS when these signals are received. If you intend to handle these signals by yourself instead of using the default set by LSLIB, you need to use the low level LSLIB function `ls_stoprex()` before the end of your signal handler.

Example: Run tasks on many machines

This example program uses `ls_rtask()` to run `rm -f /tmp/core` on user specified hosts.

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <lsf/lsf.h>
int main (int argc, char **argv)
{
    char *command[4];
    int numHosts;
    int i;
    int tid;
    if (argc <= 1) {
        printf("Usage: %s host1 [host2 ...]\n",argv[0]);
        exit(-1);
    }
    numHosts = argc - 1;
    command[0]="rm";
    command[1]="-f";
    command[2]="/tmp/core";
    command[3] = NULL;
    if (ls_initrex(numHosts, 0) < 0) {
        ls_perror("ls_initrex");
        exit(-1);
    }
    signal(SIGUSR1, SIG_IGN);
    /* Run command on the specified hosts */
    for (i=1; i<=numHosts; i++) {
        if ((tid = ls_rtask(argv[i], command, 0)) < 0) {
            fprintf(stderr, "lsrtask failed for host %s: %s\n",
                    argv[i], ls_sysmsg());
            exit(-1);
        }
        printf("Task %d started on %s\n", tid, argv[i]);
    }
    while (numHosts) {
        LS_WAIT_T status;
        tid = ls_rwait(&status, 0, NULL);
        if (tid < 0) {
            ls_perror("ls_rwait");
            exit(-1);
        }

        printf("task %d finished\n", tid);
        numHosts--;
    }
    exit(0);
}
```

The above program sets the signal handler for `SIGUSR1` to `SIG_IGN`. This causes the signal to be ignored. It uses `ls_rwait()` to poll the status of remote tasks. You could set a signal handler so that it calls `ls_rwait()` inside the signal handler.

Use the task ID to preform an operation on the task. For example, you can send a signal to a remote task explicitly by calling `ls_rkill()`.

To run the task on remote hosts one after another instead of concurrently, call `ls_rwait()` right after `ls_rtask()`.

Also note the use of `ls_sysmsg()` instead of `ls_perror()`, which does not allow flexible printing format.

The above example program produces output similar to the following:

```
% a.out hostD hostA hostB
Task 1 started on hostD
Task 2 started on hostA
Task 3 started on hostB
Task 1 finished
Task 3 finished
Task 2 finished
```

Remote tasks are run concurrently, so the order in which tasks finish is not necessarily the same as the order in which tasks are started.

Determine why job is suspended

It is frequently desirable to know the reasons why jobs are in a certain status. LSBLIB provides a function to print such information. This section describes a routine that prints out why a job is in suspending status.

lsb_suspreason()

When `lsb_readjobinfo()` reads a record of a pending job, the variables reasons and subreasons contained in the returned `struct jobInfoEnt` call `lsb_suspreason()`. This gets the reason text explaining why the job is still in pending state:

```
char *lsb_suspreason(reasons, subReasons, ld);
```

where reasons and subReasons are integer reason flags as returned by a `lsb_readjobinfo()` function while ld is a pointer to the following data structure:

```
struct loadIndexLog {
    int  nIdx;             Number of load indices configured for the
                               LSF cluster
    char **name;           List of the load index names
};
```

Call the below initialization and code fragment after `lsb_readjobinfo()` is called.

```
/* initialization */
struct loadIndexLog *indices =(struct loadIndexLog *)malloc
(sizeof(struct loadIndexLog));
char *suspreason;
/* get the list of all load index names */
indices->name = getindexlist(&indices->nIdx);
/* get and print out the suspended reason */
suspreason = lsb_suspreason(job->reasons,job-> subreasons,indices);
printf("%s\n",suspreason);
```

Determine why job is pending

Use `lsb_pendreason()` to write a program to print out the reason why a job is in pending status.

lsb_pendreason()

```
char *lsb_pendreason (int numReasons, int *rsTb,
                      struct jobInfoHead *jInfoH,
                      struct loadIndexLog *ld, int clusterId)
```

- rsTb is a reason table in which each entry contains one pending reason.
- numReasons is an integer representing the number of reasons in the table.

jobInfoHead structure

struct jobInfoHead is returned by the `lsb_openjobinfo_a()` function. It is defined as follow:

```
struct jobInfoHead {
    int   numJobs;
    LS_LONG_INT *jobIds;
    int   numHosts;    char  **hostNames;
    int   numClusters;
    char  **clusterNames;
    int   *numRemoteHosts;
    char  ***remoteHosts;};
```

ld is the same struct as used in the above `lsb_suspreason()` function call.

This program is similar but different from the above program for displaying the suspending reason. Use `lsb_openjobinfo_a()` to open the job information connection, instead of `lsb_openjobinfo()`. Because the `struct` jobInfoHead is needed as one of the arguments when calling the function `lsb_pendreason()`.

```
struct jobInfoHead *lsb_openjobinfo(jobId, jobName, user, queue, host, options);
```

The following initialization and code fragment show how to display the pending reason using `lsb_pendreason()`:

```
/* initialization */
char *pendreason;
struct loadIndexLog *indices =(struct loadIndexLog *) malloc(sizeof(struct loadIndexLog));
struct jobInfoHead *jInfoH = (struct jobInfoHead *) malloc(sizeof(struct jobInfoHead));
/* open the job information connection with mbatchd */
jInfoH = lsb_openjobinfo_a(0, NULL, user, NULL, NULL, options);
/* gets the total number of pending job, exits if failure */
if (jInfoH==NULL) {
    lsb_perror("lsb_openjobinfo");
    exit(-1);
}
/* get the list of all load index names */
indices->name = getindexlist(&indices->nIdx);
/* get and print out the pending reasons */
pendreason = lsb_pendreason(job->numReasons,job-> reasonTb,jInfoH,indices,clusterId);
printf("%s\n",pendreason);
```

Tip:

Use `ls_loadinfo()` to get the list of all load index names.

Read lsf.conf parameters

You can refer to the contents of the `lsf.conf` file or even define your own site specific variables in the `lsf.conf` file.

The `lsf.conf` file follows the Bourne shell syntax. It can be sourced by a shell script and set into your environment before starting your C program. Use these variables as environment variables in your program.

ls_readconfenv()

`ls_readconfenv()` reads the `lsf.conf` variables in your C program:

```
int ls_readconfenv(paramList, confPath)
```

where `confPath` is the directory in which the `lsf.conf` file is stored. paramList is an array of the following data structure:

```
struct config_param {
    char *paramName;       Name of the parameter, input
    char *paramValue;      Value of the parameter, output
}
```

`ls_readconfenv()` reads the values of the parameters defined in `lsf.conf` and matches the names described in the paramList array. Each resulting value is saved into the paramValue variable of the array element matching paramName. If a particular parameter mentioned in the paramList is not defined in `lsf.conf`, then on return its value is left NULL.

Example

The following example program reads the variables `LSF_CONFDIR`, `MY_PARAM1`, and `MY_PARAM2` in `lsf.conf` file and displays them on screen. Note that `LSF_CONFDIR` is a standard LSF parameter, while the other two parameters are user site specific. The example program below assumes `lsf.conf` is in `/etc` directory.

```
#include <stdio.h>
#include <lsf/lsf.h>
struct config_param myParams[] =
{
#define LSF_CONFDIR                   0
     {"LSF_CONFDIR", NULL},
#define MY_PARAM1                     1
     {"MY_PARAM1", NULL},
#define MY_PARAM2                     2
     {"MY_PARAM2", NULL},
     {NULL, NULL}
};
main()
{
    if (ls_readconfenv(myParams, "/etc") < 0) {
        ls_perror("ls_readconfenv");
        exit(-1);
    }
    if (myParams[LSF_CONFDIR].paramValue == NULL)
        printf("LSF_CONFDIR is not defined in
        /etc/lsf.conf\n");
    else
        printf("LSF_CONFDIR=%s\n",myParams[LSF_CONFDIR].paramValue);
    if (myParams[MY_PARAM1].paramValue == NULL)
        printf("MY_PARAM1 is not defined in /etc/lsf.conf\n");
    else
        printf("MY_PARAM1=%s\n", myParams[MY_PARAM1].paramValue);
    if (myParams[MY_PARAM2].paramValue == NULL)
        printf("MY_PARAM2 is not defined\n");
    else
        printf("MY_PARAM2=%s\n", myParams[MY_PARAM2].paramValue);
    exit(0);
}
```

Initialize the paramValue parameter in the config_param data structure must be initialized to `NULL`. Next, modify the paramValue to point to a result string if a matching paramName is found in the `lsf.conf` file. End the array with a `NULL` paramName.

Signal handling in Windows

LSF uses the UNIX signal mechanism to perform job control. For example, the `bkill` command in UNIX normally results in the signals SIGINT, SIGTERM, and SIGKILL being sent to the target job. Signal handling code that exists in UNIX applications allows processes to shut down in stages. In the past, the Windows equivalent to the `bkill` command was TerminateProcess(). It terminates the process immediately and does not allow the process to release shared resources the way `bkill` does.

LSF version 3.2 has been modified to provide signal notification through the Windows message queue. LSF now includes messages corresponding to common UNIX signals. This means that a customized Windows application can process these messages.

For example, the `bkill` command now sends the SIGINT and SIGTERM signals to Windows applications as job control messages. An LSF-aware Windows application can interpret these messages and shut down neatly.

To write a Windows application that takes advantage of this feature, register the specific signal messages that the application handles. Then modify the message loop to check each message before dispatching it. Take the appropriate action if the message is a job control message.

The following examples show sample code that might help you to write your own applications.

Example: Job control in a Windows application

This example program shows how a Windows application can receive a Windows job control notification from the LSF system.

Catching the notification messages involves:

- Registering the windows messages for the signals that you want to receive (in this case, `SIGTERM`).
- Look for the messages you want to catch in your GetMessage loop.

Tip:

Do not use DispatchMessage() to dispatch the message, since it is addressed to the thread, not the window. This program displays information in its main window, and waits for `SIGTERM`. Once `SIGTERM` is received, it posts a quit message and exits. A real program could do some cleanup when the `SIGTERM` message is received.

```
/* WINJCNTL.C */
#include <windows.h>
#include <stdio.h>
#define BUFSIZE 512
static UINT msgSigTerm;
static int xpos;
static int pid_ypos;
static int tid_ypos;
static int msg_ypos;
static int pid_buf_len;
static int tid_buf_len;
static int msg_buf_len;
static char pid_buf[BUFSIZE];
static char tid_buf[BUFSIZE];
static char msg_buf[BUFSIZE];
LRESULT WINAPI MainWndProc(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
    HDC hDC;
    PAINTSTRUCT ps;
    TEXTMETRIC tm;
    switch (msg) {
        case WM_CREATE:
            hDC = GetDC(hWnd);
            GetTextMetrics(hDC, &tm);
            ReleaseDC(hWnd, hDC);
            xpos = 0;
            pid_ypos = 0;
            tid_ypos = pid_ypos + tm.tmHeight;
            msg_ypos = tid_ypos + tm.tmHeight;
            break;
        case WM_PAINT:
            hDC = BeginPaint(hWnd, &ps);
            TextOut(hDC, xpos, pid_ypos, pid_buf, pid_buf_len);
            TextOut(hDC, xpos, tid_ypos, tid_buf, tid_buf_len);
            TextOut(hDC, xpos, msg_ypos, msg_buf, msg_buf_len);
            EndPaint(hWnd, &ps);
            break;
        case WM_DESTROY:
            PostQuitMessage(0);
            break;
        default:
            return DefWindowProc(hWnd, msg, wParam, lParam);
    }
    return 0;
}
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
LPSTR lpCmdLine, int nCmdShow)
{
    ATOM rc;
    WNDCLASS wc;
    HWND hWnd;
    MSG msg;
/* Create and register a windows class */
    if (hPrevInstance == NULL) {
            wc.style = CS_OWNDC | CS_VREDRAW | CS_HREDRAW;
            wc.lpfnWndProc = MainWndProc;
            wc.cbClsExtra = 0;
            wc.cbWndExtra = 0;
            wc.hInstance = hInstance;
            wc.hIcon = LoadIcon(NULL, IDI_APPLICATION);
            wc.hCursor = LoadCursor(NULL, IDC_ARROW);
            wc.hbrBackground = (HBRUSH) (COLOR_WINDOW + 1);
            rc = RegisterClass(&wc);
    }
/* Register the message we want to catch */
    msgSigTerm = RegisterWindowMessage("SIGTERM");
/* Format some output for the main window */
sprintf(pid_buf, "My process ID is: %d", GetCurrentProcessId());
pid_buf_len = strlen(pid_buf);
sprintf(tid_buf, "My thread ID is: %d", GetCurrentThreadId());
tid_buf_len = strlen(tid_buf);
sprintf(msg_buf, "Message ID is: %u", msgSigTerm);
msg_buf_len = strlen(msg_buf);
/* Create the main window */
    hWnd = CreateWindow("WinJCntlClass",
            "Windows Job Control Demo App",
            WS_OVERLAPPEDWINDOW,
            0,
            0,
            CW_USEDEFAULT,
            CW_USEDEFAULT,
            NULL,
            NULL,
            hInstance,
            NULL);
    ShowWindow(hWnd, nCmdShow);
/* Enter the message loop, waiting for msgSigTerm. When we get
it, just post a quit message */
    while (GetMessage(&msg, NULL, 0, 0)) {
        if (msg.message == msgSigTerm) {
            PostQuitMessage(0);
        } else {
            TranslateMessage(&msg);
            DispatchMessage(&msg);
        }
    }
    return msg.wParam;
}
```

Job control in a console application

Example

This example program shows how a console application can receive a Windows job control notification from the LSF system.

Catching the notification messages involves:

- Registering the windows messages for the signals that you want to receive (in this case, `SIGINT` and `SIGTERM`).
- Creating a message queue by calling PeekMessage (this is how Microsoft suggests console applications should create message queues).
- Look for the message you want to catch enter a GetMessage loop.

Tip:

Do not DispatchMessage here, since you do not have a window to dispatch to.

This program sits in the message loop. It is waiting for `SIGINT` and `SIGTERM`, and displays messages when those signals are received. A real application would do clean-up and exit if it received either of these signals.

```
/* CONJCNTL.C */
#include <windows.h>
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    DWORD pid = GetCurrentProcessId();
    DWORD tid = GetCurrentThreadId();
    UINT msgSigInt = RegisterWindowMessage("SIGINT");
    UINT msgSigTerm = RegisterWindowMessage("SIGTERM");
    MSG msg;
/* Make a message queue -- this is the method suggested by MS */
    PeekMessage(&msg, NULL, WM_USER, WM_USER, PM_NOREMOVE);
    printf("My process id: %d\n", pid);
    printf("My thread id: %d\n", tid);
    printf("SIGINT message id: %d\n", msgSigInt);
    printf("SIGTERM message id: %d\n", msgSigTerm);
    printf("Entering loop...\n");
    fflush(stdout);
    while (GetMessage(&msg, NULL, 0, 0)) {
        printf("Received message: %d\n", msg.message);
        if (msg.message == msgSigInt) {
            printf("SIGINT received, continuing.\n");
        } else if (msg.message == msgSigTerm) {
            printf("SIGTERM received, continuing.\n");
        }
        fflush(stdout);
    }
    printf("Exiting.\n");
    fflush(stdout);
    return EXIT_SUCCESS;
}
```

CHAPTER

5

User-Level Checkpointing

User-level checkpointing

LSF provides a method to checkpoint jobs on systems that do not support kernel-level checkpointing called user-level checkpointing. To implement user-level checkpointing, you must have access to your applications object files (.o files), and they must be re-linked with a set of libraries provided by LSF. This approach is transparent to your application, its code does not have to be changed and the application does not know that a checkpoint and restart has occurred.

By default, the checkpoint libraries are installed in LSF_LIBDIR and `echkpnt` and `erestart` are installed in the LSF_SERVERDIR.

Optionally, third party checkpoint and restart implementations can be used with LSF. You must use the `echkpnt` and `erestart` supplied with the implementations. To avoid overwriting the `echkpnt` and `erestart` supplied by LSF, install any third party implementations in a separate directory by defining LSB_ECHKPNT_METHOD and LSB_ECHKPNT_METHOD_DIR as environment variables or in `lsf.conf`.

Limitations

There are restrictions to the use of the current implementation of the checkpoint library for user-level checkpointing. These are:

- The checkpointed process can only be restarted on hosts of the same architecture and with the same operating system as the host on which the checkpoint was created.
- Only single process jobs can be checkpointed.
- Processes with open pipes and sockets can be checkpointed but may not properly restart as the pipes and sockets are not re-opened on restart.
- If a process has `stdin`, `stdout`, or `stderr` as open pipes, all data in the pipes is lost on restart.
- The checkpointed process cannot be operating on a private stack when the checkpoint happens.
- The checkpointed process cannot use internal timers.
- The checkpointed program must be statically linked.

  `SIGHUP` is used internally to implement checkpointing. Do not use this signal in programs to be checkpointed.

User-level checkpointable jobs

Building a user-level checkpointable job involves re-linking your application object files (.o files) with the LSF checkpoint startup routine and library. LSF also provides a set of replacement linkers that call the standard linkers on your platform with the correct options to build a checkpointable application. LSF provides:

- `libckpt.a`, the checkpoint library
- `ckpt_crt0.o`, the checkpoint startup routine
- `ckpt_ld` the checkpoint linker for C language applications
- `ckpt_ld_f` the checkpoint linker for Fortran applications

Library

The checkpoint library replaces low-level system calls such as open(), close(), and dup(), and contains signal handlers and routines to internally implement checkpointing.

Startup routine

The startup routine replaces the language-level module that calls main(), sets the checkpoint signal handler, and initializes internal data structures used to record job information.

Linkers

The checkpoint linkers are used to re-link your application with the checkpoint library and startup routine. They are shell scripts that call the standard linkers on your operating system with the correct options. The scripts are designed to use the native compilers on most platforms. Use `ckpt_ld` for C language applications and `ckpt_ld_f` for Fortran applications. The following compilers are supported by the `ckpt_ld` replacement linker:

Operating System	Compiler
AIX	`cc`
HP-UX	`c89`

IRIX 6.2	For IRIX 6.2 you need to use `cc` with the `-non_shared -mips2 -32` compiler options, and `ckpt_ld` with `-mips2 -32` linker options. For example, to compile and link `my_job.c`:
	`% cc -c my_job.c -non_shared -mips2 -32`
	`% ckpt_ld -o my_job my_job.o -mips2 -32`

OSF1	`cc`
Solaris	`cc` (SUN C compiler) and `gcc`
SunOS	`gcc`

Re-Link user-level applications

To re-link your application, you must have access to the object files (.o files) for your application.

1. If you are using third party applications, the vendor must supply you with the object files.
2. If you are building your own applications you need to first compile them without linking.

   C++ applications need to be modified before re-linking.

C Language applications

- To compile a C language application without linking, run the compiler with the `-c` option instead of the `-o` option. For example, to compile an object file for `my_job`:

```
% cc -c my_job.c
```

- To re-link a C language object file use the supplied LSF replacement linker `ckpt_ld`. For example, to re-link an object file for an application called `my_job`:

```
% ckpt_ld -o my_job my_job.o
```

Fortran applications

- To compile a Fortran application without linking, run the compiler with the `-c` option instead of the `-o` option. For example, to compile an object file for `my_job`:

```
% f77 -c my_job.f
```

- To re-link a Fortran object file use the supplied LSF replacement linker `ckpt_ld_f`. For example, to re-link an object file for an application called `my_job`:

```
% ckpt_ld_f -o my_job my_job.o
```

Troubleshoot user-level re-linking

If an error is reported when using `ckpt_ld` to link your application with the checkpoint libraries:

1. Follow the troubleshooting steps to isolate the problem.
2. If you cannot resolve your errors, call Platform Customer Support.

Note:

The `ckpt_ld` replacement linker is designed for C language applications, if your application was created using C++, you need to modify your files before re-linking.

Replacement linkers

The replacement linkers are shell scripts designed to use the standard compilers on your OS with the correct options to build a checkpointable executable.

The linkers do the following:

- Include the startup routine by replacing the module that calls `main()` with `ckpt_crt0.o`
- Include the checkpoint library by adding `libckpt.a`
- Force as much static linking as possible

Resolve re-linking errors

To resolve linking errors, you need to step through the linking process performed by the linker. To do this, perform the following procedures:

1. View the linking script
2. Include the startup library
3. Include the checkpoint library
4. Force static linking

View the linking script

- View the low-level linking script by running your linker in verbose mode.

  This will display the libraries called by your linker. Use this information to help determine which files need to be replaced.

  Refer to the man page supplied with your compiler to determine the verbose mode switch. The following table lists the verbose mode switch for some operating systems.

Operating System	Verbose Mode Switch
SUNOS/Solaris	-#
AIX	-v
IRIX	-show -non_shared
HP-UX	-v
OSF1	-v -non_shared

For example, running the Sparc C Compiler 3.0 with the verbose switch, `-#`, for `my_job.o`:

```
% cc -o -# my_job my_job.o
```

```
/usr/ccs/bin/ld /opt/SUNWspro/SC3.0/lib/crti.o /opt/SUNWspro/SC3.0/lib/
crt1.o /opt/SUNWspro/SC3.0/lib/__fstd.o /opt/SUNWspro/SC3.0/lib/values-
xt.o -o my_job my_job.o -Y P,/opt/SUNWspro/SC3.0/lib:/usr/ccs/lib:/usr/
lib -Qy -lc /opt/SUNWspro/SC3.0/lib/crtn.o
```

Include the startup library

Add the startup library:

- Replace the library that calls `main()` with `ckp_crt0.o`. To determine which library calls `main()`, run `nm` for all libraries listed in the low-level linking script. For example:

```
% nm /opt/SUNWspro/SC3.0/lib/crt1.o | grep -i main
```

Replace /opt/SUNWspro/SC3.0/lib/crt1.o with /usr/share/lsf/lib/ckpt_crt0.o:

```
/usr/ccs/bin/ld /opt/SUNWspro/SC3.0/lib/crti.o /usr/share/lsf/lib/
ckpt_crt0.o /opt/SUNWspro/SC3.0/lib/__fstd.o /opt/SUNWspro/SC3.0/lib/
values-xt.o -o my_job my_job.o -Y P,/opt/SUNWspro/SC3.0/lib:/usr/ccs/lib:/
usr/lib -Qy -lc /opt/SUNWspro/SC3.0/lib/crtn.o
```

Include the checkpoint library

- Add `libckpt.a` after language-specific libraries and before system-specific libraries. For example:

```
/usr/ccs/bin/ld /opt/SUNWspro/SC3.0/lib/crti.o /usr/share/lsf/lib/
ckpt_crt0.o /opt/SUNWspro/SC3.0/lib/__fstd.o /opt/SUNWspro/SC3.0/lib/
values-xt.o -o my_job my_job.o /usr/share/lsf/lib/libckpt.a -Y P,/opt/
SUNWspro/SC3.0/lib:/usr/ccs/lib:/usr/lib -Qy -lc /opt/SUNWspro/SC3.0/lib/
crtn.o
```

Force static linking

- Force your application to link statically to as many libraries as possible.

  Refer to the documentation supplied with your compiler for more information about static linking. For example, on Solaris the `-Bstatic` and `-Bdynamic` compiler switches are used to force modules to statically link wherever possible:

```
/usr/ccs/bin/ld -Bstatic /opt/SUNWspro/SC3.0/lib/crti.o /usr/share/lsf/
lib/ckpt_crt0.o /opt/SUNWspro/SC3.0/lib/__fstd.o /opt/SUNWspro/SC3.0/lib/
values-xt.o -o my_job my_job.o /usr/share/lsf/lib/libckpt.a -Y P,/opt/
SUNWspro/SC3.0/lib:/usr/ccs/lib:/usr/lib -Qy -lc -Bdynamic -ldl -Bstatic /
opt/SUNWspro/SC3.0/lib/crtn.o
```

Re-Link C++ applications

To use the replacement linker on C++ applications, the module that calls `main()` must be extracted from its library file and included in the linking script.

The following example `Verilog` application is written in C++ and being re-linked on Solaris. It reports an undefined symbol `main` in `libckpt.a`:

```
/usr/ccs/bin/ld /opt/SUNWspro/SC3.0.1/lib/crti.o /opt/SUNWspro/SC3.0.1/lib/crt1.o /opt/SUNWspro/
SC3.0.1/lib/cg89/__fstd.o /opt/SUNWspro/SC3.0.1/lib/values-xt.o -Y P,lxx/lib:opt/SUNWspro/SC3.0.1/
lib:/usr/ccs/lib:/usr/lib -o verilog verilog.o verilog/lib/*.o lib/libcman.a -L/usr/openwin/lib -
lXt -X11 lib/libvoids.a -lm -lgen lxx/lib/_main.o -lC -lC_mtstubs -lsocket -lnsl -lintl -w -c -
ldl /opt/SUNWspro/lib/crtn.o
```

1. To determine which library contains `main()`, run `nm` for all libraries listed in the low-level linking script. For example:

   ```
   % nm lib/libvoids.a | grep main
   ```

2. This module must be extracted using:

   ```
   % ar x lib/libvoids.a main.o
   ```

3. The `main.o` object file must be included in the re-linking script to generate a checkpointable executable:

   ```
   /usr/ccs/bin/ld /opt/SUNWspro/SC3.0.1/lib/crti.o /opt/SUNWspro/SC3.0.1/lib/crt1.o /opt/
   SUNWspro/SC3.0.1/lib/cg89/__fstd.o /opt/SUNWspro/SC3.0.1/lib/values-xt.o -Y P,lxx/lib:opt/
   SUNWspro/SC3.0.1/lib:/usr/ccs/lib:/usr/lib -o verilog main.o verilog.o verilog/lib/*.o lib/
   libcman.a -L/usr/openwin/lib -lXt -X11 lib/libvoids.a -lm -lgen lxx/lib/_main.o -lC -lC_mtstubs
   -lsocket -lnsl -lintl -w -c -ldl /opt/SUNWspro/lib/crtn.o
   ```

CHAPTER

6

External Scheduler Plugins

About external scheduler plugins

The default scheduler plugin modules provided by LSF may not satisfy all the particular scheduling policies you need. You can use the LSF scheduler plugin API to customize existing scheduling policies or implement new ones that can operate with existing LSF scheduler plugin modules.

- Certain scheduling policies can be implemented based on the specific requirements of your site.
- Customized policies can be incorporated with other LSF features to provide seamless behavior. Your custom scheduling policy can influence, modify, or override LSF scheduling decisions.
- Your plugin can take advantage of the load and host information already maintained by LSF.
- The scheduler plugin architecture is fully external and modular; new scheduling policies can be prototyped and deployed without having to change the compiled code of LSF.

Sample plugin code

Sample code for an example external scheduler plugin, and information about writing, building, and configuring your own custom scheduler plugin is located in:

```
LSF_TOP/7/misc/examples/external_plugin/
```

Write an external scheduler plugin

Scheduling policies can be applied into two phases of a scheduling cycle: match phase and allocation phase.

Match/sort phase

In match phase, scheduler prepares candidate hosts for jobs. All jobs with the same resource requirements share the same candidate hosts. The plugin at this phase can decide which host is eligible for future consideration. If the host is not eligible for the job, it is removed from the candidate host list. At the same time, the plugin associates a pending reason with the removed host, which will be shown by the `bjobs` command.

Finally, the plugin can decide which candidate host should be considered first in future.

The plugin in this phase provides two functions:

Match():

Doing filtering on candidate hosts

Sort():

Doing ordering on candidate hosts

Input and output of match phase

The input/output of this phase are candHostGroupList and PendingReasonTable. Candidate hosts are divided into several groups. Jobs can only use hosts from one of candHostGroup in the candHostGroupList.

The plugin filters the candHostGroups in candHostGroupList, removes the ineligible hosts from the group, and sets the pending reason in the PendingReasonTable.

Plugin Invocation

Since each plugin does match/sort based on certain resource requirements, it decides which host is qualified and which should be first based on certain kinds of resource requirements. The scheduler organizes the Match() and Sort() into the handler of each resource requirement.

After the handler is created, all that plugin needs to do is to register it to scheduler framework. Then it is the scheduler framework's responsibility to call each handler doing match and sort and handling each specific resource requirement.

When the plugin registers the handler, a resource criteria type is associated with the handler. The Criteria Type indicates which kind of resource requirement the handler is handling.

Handler functions

Together with Match() and Sort(), there are other two handler functions:

New()

Gets the user-specific resource requirements string, parses it, creates the handler-specific data, and finally attaches the data to related resource requirement.

Free()

Frees the handler-specific data when not needed.

See `sched_api.h` for details.

Implement match phase

See `sch.mod.matchexample.c` for details.

1. Define resource criteria type, handler-specific data, and user specific pending reason, as required.
2. Implement handler functions.
3. Implement initialization functions.

Step 1.

- Define resource criteria type, handler-specific data, and user specific pending reason.

  The criteria type indicates the kind of resource requirement the handler is handling. Usually, the external plugin handler only handles external resource requirement (string) which is specified through `bsub` command using the `-extsched` option.

  In order to use `-extsched`, you must set LSF_ENABLE_EXTSCHEDULER=y in `lsf.conf`.

  New() function parses the external resource requirement string, and stores the parsed resource to handler-specific data.

  handler-specific data is a container used to store any data which is needed by the handler.

  If the plugin needs to set a user specific pending reason, a pending reason ID needs to be defined. See lsb_reason_set() in `sched_api.h` for more information.

Step 2.

Implement handler functions: New(), Free(), Match(), and Sort().

1. New():
   a) Get external resource requirement message (lsb_resreq_getextresreq()).
   b) Find my message, and parse it.
   c) Create handler-specific data, and store parsing result in it.
   d) Create a key, (in example, just use external message as a key).
   e) Attach the handler-specific data (lsb_resreq_setobject()).
2. Free():

   Free whatever in handler-specific data.
3. Match(): (handler-specific data is passed in)
   a) Go through all candidate host groups.(lsb_cand_getnextgroup())
   b) Look at candidate host in each group. If a host is not eligible, remove it from group and set pending reason (lsb_cand_removehost(), lsb_reason_set()).
4. Sort(): (handler-specific data is passed in)
   a) Go through all candidate host groups (lsb_cand_getnextgroup()).
   b) Sort the candidate hosts in the group.

Step 3.

1. Implement sched_init().

This function is the plugin initialization function, which is called when the plugin is loaded.

2. Create handler, and register it to scheduler framework (lsb_resreq_registerhandler).

Allocation phase

In allocation phase, the scheduler makes allocation decisions for each job. It assigns host slot, memory, and other resources to the job. It also checks if the allocation satisfies all constrains defined in configuration, such as queue slot limit, deadline for the job, etc.

Your plugin at this phase can modify allocation decisions made by another LSF module.

Limitations or allocation modifications

1. External plugin is only allowed to change the host slot distribution, i.e., reduce/ increase the slot usage on certain host, add more hosts to the allocation. Other resource usage modification is not supported now.
2. External plugin is not allowed to remove a host from an allocation.
3. External plugin cannot change reservation in an allocation.

Input and output of allocation phase

INPUT:

job: current job we are making allocation for.

candHostGroupList: (see section 2.1.1)

pendingReasonTable: (see section 2.1.1)

INPUT/OUTPUT:

alloc: LSF allocation decision is passed in, and plugin will modify it, and make its own allocation decision on top of it.

Invocation

At allocation phase, the plugin needs to provide a callback function, AllocatorFn, which adjusts allocation decisions made by LSF. This function must be registered to the scheduler framework. The scheduler framework calls it after LSF makes a decision for the job.

In addition to AllocatorFn(), the plugin may also need to provide a New() function in the handler for the user-specific resource criteria, if there are any. If there is no such user-specific resource requirement, AllocatorFn() is applied to all jobs.

Allocation phase

See `sch.mod.allocexample.c` for details..

1. Optional.

   Define criteria type for external resource requirements.
2. Optional.

   Implement New() function in the handler for the resource criteria type.
3. Implement callback AllocatorFn():
   a) Check if the allocation has the type of SCH_MOD_DECISION_DISPATCH. If not, just return (lsb_alloc_type()).

   b) Optional. Get external message, and decide whether to continue (lsb_job_getextresreq()).
   c) Get current slot distribution in allocation and availability information for all candidate hosts (lsb_alloc_gethostslot()).
   d) Modify the allocation (lsb_alloc_modify()).

      Use lsb_alloc_modify() gradually, not for big changes, because lsb_alloc_modify() may return FALSE due to conflict with other scheduling policies, such as user slot limits on host.

   In `sch.mod.allocexample.c`, slots are adjusted in small steps.

4. Implement sched_init(). This function is the plugin initialization function, which is called when the plugin is loaded.
   a) Optional. Create a handler for resource requirement processing, and register it to the scheduler framework (lsb_resreq_registerhandler()).
   b) Register the allocation callback AllocatorFn() (lsb_alloc_registerallocator()).

Build the external scheduler plugin.

1. Set INCDIR and LIBDIR in the makefile to point to the appropriate directories for the LSF include files and libraries.
2. Create a `Make.def` for the platform on which you want to build the plugin. The `Make.def` should be located in the LSF_MISC directory at the same level of `Make.misc`.

   All `Make.def` templates for each platform are in `config` directory. For example, if you want run examples on Solaris2.6, use following command to create `Make.def`:

   ln -s config/Make.def.sparc-sol2 Make.def

   You can also change the file, if necessary.
3. Run `make` in current directory.

Enable and Use the external scheduler plugin

Use `sch.mod.matchexample.c` as an example.

1. Copy `schmod_matchexample.so` to LSF_LIBDIR (defined in lsf.conf).
2. Configure the plugin in `lsb.modules`; add following line after all LSF modules:

   ```
   schmod_matchexample     ()          ()
   ```
3. `badmin mbdrestart`
4. Use `bsub` to submit a job.

   If external message is needed, use the option `-extsched`.

   For example:

   ```
   bsub -n 2 -extsched "EXAMPLE_MATCH_OPTIONS=goedel" -R "type==any" sleep 1000
   ```

   In order to use `-extsched`, you must set LSF_ENABLE_EXTSCHEDULER=y in `lsf.conf`.
5. Use `bjobs` to look at external message, and customized pending reason.

   ```
   --------------------------------------------------------------------------
   ./bjobs -lp
   Job <224>, User <yhu>, Project <default>, Status <PEND>, Queue <short>, Job Pri
                        ority <500>, Command <sleep 1000>
   Thu Nov 29 15:08:05: Submitted from host <goedel> with hold, CWD <$HOME/LSF4_1/
                        utopia/lsbatch/cmd>, Requested Resources <type==any>;
    PENDING REASONS:
    Load information unavailable: pauli, varley, peano, bongo;
    Closed by LSF administrator: curie, togni;
    Customized pending reason number 20002: goedel;
   ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    SCHEDULING PARAMETERS:
              r15s   r1m  r15m   ut      pg    io   ls    it    tmp    swp    mem
    loadSched   -     -     -     -       -     -    -     -     -      -      -
    loadStop    -     -     -     -       -     -    -     -     -      -      -
             total_jobs mbd_size
    loadSched        -        -
    loadStop         -        -
    EXTERNAL MESSAGES:
    MSG_ID FROM       POST_TIME      MESSAGE                             ATTACHMENT
    0          -             -                        -                      -
    1      yhu        Nov 29 15:08   EXAMPLE_MATCH_OPTIONS=goedel            N
                                    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
   ---------------------------------------------------------------------------
   ```

Scheduler API reference summary

See the following API man pages for details:

- AllocatorFn.3
- RsrcReqHandler_FreeFn.3
- RsrcReqHandler_MatchFn.3
- RsrcReqHandler_NewFn.3
- RsrcReqHandler_SortFn.3
- _RsrcReqHandlerType.3
- candHost.3
- candHostGroup.3
- hostSlot.3
- lsb_alloc_gethostslot.3
- lsb_alloc_modify.3
- lsb_alloc_registerallocator.3
- lsb_alloc_type.3
- lsb_cand_getavailslot.3
- lsb_cand_getnextgroup.3
- lsb_cand_removehost.3
- lsb_job_getaskedslot.3
- lsb_job_getextresreq.3
- lsb_job_getrsrcreqobject.3
- lsb_reason_set.3
- lsb_resreq_getextresreq.3
- lsb_resreq_registerhandler.3
- lsb_resreq_setobject.3

Debug the external scheduling plugin

1. `mbschd.log.goedel` will show which plugins are successfully loaded. If loading fails, the error message is also logged.
2. Use debug tool to debug plugins, such `gdb`, `dbx`, etc. Attach to `mbschd`, and set breakpoint in the functions of plugin.

APPENDIX

A

Tutorials

Simple batch job

```
/******************************************************
* LSBLIB -- Examples
*
* lsb_submit()
* Submit command as an lsbatch job using the simplest
* version of lsb_submit()
* Note: there is no error checking in this program.
******************************************************/
#include <lsf/lsbatch.h>
/* Use the header file lsbatch.h when writing programs that use the LSF API. */
#include "combine_arg.h"
/* To use the function "combine_arg" to combine arguments on the command line include its header
file "combine_arg.h". */
int main(int argc, char **argv)
{
int i;
struct submit req;
/* req holds the job specification. */
memset(&req, 0, sizeof(req));
/* initializes req to avoid core dump */
struct submitReply  reply;
/* reply holds the result of submission. */
lsb_init(argv[0]);
/* Before using any batch library function, call lsb_init(). lsb_init() initializes the
configuration environment. */
/* Set up the job’s specifications by initializing some of the flags in lsb_submit(). */
req.options = 0;
req.options2 = 0;
/* Set options and options2 to 0 to indicate that no options are selected. options are used by
lsb_submit() to indicate modifications to the job submission action to be taken.*/
for (i = 0; i < LSF_RLIM_NLIMITS; i++)
     req.rLimits[i] =           DEFAULT_RLIMIT;
/* Initialize resource limits to default limits (no limit).*/
req.numProcessors = 1;
req.maxNumProcessors = 1;
/* Initialize the initial number and the maximum number of processors needed by a (parallel) job.*/
req.beginTime = 0;
/* To dispatch a job without delay assign 0 to beginTime..
req.termTime  = 0;
To have no terminating deadlines, assign 0 to termTime. */
req.command = combine_arg(argc,argv);
/* Initialize the command line by assigning combine_arg to command. */
lsb_submit(&req, &reply);
/*Call lsb_submit() to submit the job with specifications. */
    exit(0);
} /* main */
```

Batch job with error checking

```
/******************************************************
* LSBLIB -- Examples
*
* lsb_submit()
* Use lsb_submit() in the simplest way with error
* checking
*****************************************************/
#include <stdlib.h>
#include <stdio.h>
#include <lsf/lsbatch.h>
#include "combine_arg.h"
    /* To use the function "combine_arg" to combine arguments on the         command line include
its header file "combine_arg.h". */
int main(int argc, char **argv)
{
    int i;
    struct submit req;            /* job specifications */
    memset(&req, 0, sizeof(req)); /* initializes req */
    struct submitReply  reply;    /* results of job submission */
    int  jobId;                    /* job ID of submitted job
*/
/* Check the return value of lsb_init() to ensure that the initialization of LSBLIB is
successful. */
if (lsb_init(argv[0]) < 0) {
sb_perror("simbsub: lsb_init() failed");
  l    exit(-1);
 }
/* Check if the input is in the correct format: "./simbsub COMMAND [ARGUMENTS]"
(simbsub is the name of this executable program). */
if (argc < 2) {
 fprintf(stderr, "Usage: simbsub command\n");
 exit(-1);
  }
    req.options = 0;    /* Set options and options2 to 0 */
    req.options2 = 0;    /* to indicate no options are selected */
    req.beginTime = 0;  /* Set beginTime to 0 to dispatch job                            without
delay */
    req.termTime  = 0;  /* Set termTime to 0 to indicate no                          terminating
deadline */
/* Set Resource limits to default*/
for (i = 0; i < LSF_RLIM_NLIMITS; i++)
     req.rLimits[i] = DEFAULT_RLIMIT;
/*Initialize the initial number and maximum number of processors needed by a (parallel) job*/
req.numProcessors = 1;
req.maxNumProcessors = 1;
   req.command = combine_arg(argc,argv);   /* Initialize
command line of                                               job */
printf("---------------------------------------------\n");
   jobId = lsb_submit(&req, &reply);      /*submit the job
with                                                specifications */
    exit(0);
} /* main */
```

Batch Job with lsb_submit()

```
/******************************************************
* LSBLIB -- Examples
```

```
*
* lsb_submit() usage that is equivalent to "bsub" command * with no options
*****************************************************/
#include <stdlib.h>
#include <stdio.h>
#include <lsf/lsbatch.h>
#include "combine_arg.h"
    /* To use the function "combine_arg" to combine arguments on the         command line include
its header file "combine_arg.h". */
int main(int argc, char **argv)
{
    int i;
    struct submit req;          /* job specifications */
    memset(&req, 0, sizeof(req)); /* initializes req */
    struct submitReply  reply;  /* results of job submission */
    int  jobId;                 /* job ID of submitted job */
    /* initialize LSBLIB  and  get  the  configuration environment */
    if (lsb_init(argv[0]) < 0) {
        lsb_perror("simbsub: lsb_init() failed");
        exit(-1);
    }
    /* check if input is in the right format: "./simbsub COMMAND ARGUMENTS" */
    if (argc < 2) {
    fprintf(stderr, "Usage: simbsub command\n");
    exit(-1);
    }
/ * In order to synchronize the job specification in lsb_submit() to the default used by bsub,
the following variables are defined. (By default, bsub runs the job in 1 processor with no
resource limit.) */
/*Resource limits are initialized to default limits (no limit).*/
for (i = 0; i < LSF_RLIM_NLIMITS; i++)
     req.rLimits[i] = DEFAULT_RLIMIT;
/* Initialize the initial number and maximum number of processors needed by a (parallel) job. */
req.numProcessors = 1;
req.maxNumProcessors = 1;
    req.options = 0;   /* Set options and options2 to 0 */
    req.options2 = 0;  /* Select no options is selected */
    req.beginTime = 0;  /* Dispatch job without delay */
    req.termTime = 0;  /* Use no terminating deadline */
    req.command = combine_arg(argc,argv);  /* job command line */
    printf("-----------------------------------------------\n");
    jobId = lsb_submit(&req, &reply);  /* submit the job with
specifications */
    if (jobId < 0) /* if job submission fails, lsb_submit                      returns -1 */
switch (lsberrno) { /* and sets lsberrno to indicate the error */
    case LSBE_QUEUE_USE:
    case LSBE_QUEUE_CLOSED:
        lsb_perror(reply.queue);
        exit(-1);
    default:
        lsb_perror(NULL);
        exit(-1);
    }
    exit(0);
} /* main */
```

Batch job for a specific queue

```
/*****************************************************
* LSBLIB -- Examples
*
* bsub -q
* This program is equivalent to using the "bsub -q queue_name" * command
*****************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <lsf/lsbatch.h>
#include "combine_arg.h"
```

```
    /* To use the function "combine_arg" to combine arguments on the         command line include
its header file "combine_arg.h". */
int main(int argc, char **argv)
{
    int i;
    struct submit req;         /* job specifications */
    memset(&req, 0, sizeof(req)); /* initializes req */
    struct submitReply  reply;  /* results of job submission */
    int  jobId;                 /* job ID of submitted job */
/* Initialize LSBLIB and get the configuration environment */
    if (lsb_init(argv[0]) < 0) {
        lsb_perror("simbsub: lsb_init() failed");
        exit(-1);
    }
/*Check if input is in the right format: "./simbsub COMMAND   ARGUMENTS" */
    if (argc < 2) {
      fprintf(stderr, "Usage: simbsub command\n");
exit(-1);
    }
req.options |= SUB_QUEUE;
/* SUB_QUEUE indicates that the job is dispatched to a specific queue. */
req.queue="normal";
/* Queue name is given by user (e.g. "normal")
The queue name has to be valid (check the queue using bqueues) */
    req.options2 = 0;
    for (i = 0; i < LSF_RLIM_NLIMITS; i++)    /* resources limits */
        req.rLimits[i] = DEFAULT_RLIMIT;
    req.beginTime = 0; /* specific dispatch date and time */
    req.termTime  = 0; /* specifies job termination deadline */
    req.numProcessors = 1; /* initial num of processors needed by a (parallel) job */
    req.maxNumProcessors = 1; /*max num of processors required to run the parallel job */
    req.command = combine_arg(argc,argv); /* command line of job */
    jobId = lsb_submit(&req, &reply);   /* submit the job with specifications */
    if (jobId < 0)      /* if job submission fails, lsb_submit returns -1 */
switch (lsberrno) {    /* and sets lsberrno to indicate the error */
case LSBE_QUEUE_USE:
case LSBE_QUEUE_CLOSED:
    lsb_perror(reply.queue);
    exit(-1);
default:
    lsb_perror(NULL);
    exit(-1);}
    exit(0);
}
    /* main */
```

Supplementary files

```
/* combine_arg.h */
#include <stdlib.h>
#include <string.h>
/* combine_arg.h */
char *combine_arg(int c,char **arg); /* combine the arguments on command line */
/* combine_arg.c */
/* combine the arguments on command line */
#include "combine_arg.h"
char *combine_arg(int c,char **arg)
{
    int i,j=0;
    char *s;
    /* counts the number of characters in the arguments */
    for (i=1;i<c;i++)
        j+=strlen(arg[i])+1;
    /* paste the arguments */
    s = (char *)malloc(j*sizeof(char));
    memset (s, "\0", sizeof(s));
    strcat(s,arg[1]);
    for (i=2;i<c;i++)
    {
        strcat(s," ");
        strcat(s,arg[i]);
    }
    return s;
}
/* submit_cmd.h */
#include <lsf/lsbatch.h>
#include "combine_arg.h"
int submit_cmd(struct submit *req, struct submitReply *reply, int c, char **arg);
/* submit_cmd.c */
/* submit a job with specifications (without error checking) */
#include "submit_cmd.h"
int submit_cmd(struct submit *req, struct submitReply *reply, int c, char **arg)
{
    int  i;
    lsb_init(arg[0]);
    for (i = 0; i < LSF_RLIM_NLIMITS; i++)
        req->rLimits[i] = DEFAULT_RLIMIT;
    req->numProcessors = 1;
    req->maxNumProcessors = 1;
    req->options = 0;
    req->options2 = 0;
    req->command = combine_arg(c,arg);
    req->beginTime = 0;
    req->termTime  = 0;
    return lsb_submit(req, reply);
}
```

APPENDIX

B

Common LSF Functions

Job related functions

Delete a job

To delete a job, send a `KILL` signal to the job by using `lsb_signaljob()` or use `lsb_deletejob()` to kill the job.

```
int lsb_deletejob(jobId, times, options)
LS_LONG_INT jobId;
int times;
int options;Set to 0
```

`lsb_deletejob()` deletes the job after a specific number of runs. The variable times represents the number of runs .

View job output

The output from an LSF job is normally not available until the job is finished. However, LSBLIB provides `lsb_peekjob()` to retrieve the name of a job file for the job specified by jobId.

To get the job output and job error files, append `.out` or `.err` to the end of the base job file name from `lsb_peekjob()`.

Only the job owner can use `lsb_peekjob()` to see job output.

```
char *lsb_peekjob(jobId)
LS_LONG_INT  jobId;                   Job ID
```

On success, the job file name is returned. On failure, it returns `NULL` and sets `lsberrno` to indicate the error.

The next call reuses the storage for the file name.

Move jobs from one host to another

Use `lsb_mig()` to migrate a job from one host to another.

```
int lsb_mig(mig, badHostIdx);
struct submig *mig;           Job to be migrated
int    *badHostIdx;
```

If the call fails, (**askedHosts)[*badHostIdx] is not a host known to the LSF system.

`lsf.batch.h` defines the `struct submig` to hold the details of the job to be migrated. It has the following fields:

```
struct submig {
    LS_LONG_INT jobId;           Job ID to be migrated
    int         options;
    int         numAskedHosts;    Number of hosts supplied for migration
    char        **askedHosts;     Array of pointers to the hosts
};
```

For the values of options, see the options field of `struct submit` used in `lsb_submit()` function call.

On success, `lsb_mig()` returns 0. On failure, it returns -1 and sets `lsberrno` to the usual error.

External job message and data exchange

`lsb_postjobmsg()` sends an external message/status to a job. It can also transfer an attached data file through a TCP connection. The posted messages and attached data files can be read from `mbatchd` by invoking `lsb_readjobmsg()`.

```
int lsb_postjobmsg(jobExternalMsgReq, fileName)
struct jobExternalMsgReq *jobExternalMsgReq;
    char *fileName;           Data file to be attached
int lsb_readjobmsg(jobExternalMsgReq, jobExternalMsgReply)
struct jobExternalMsgReq *jobExternalMsgReq;
struct jobExternalMsgReply *jobExternalMsgReply;
```

Use struct jobExternalMsgReq as a parameter in both `lsb_postjobmsg()` and `lsb_readjobmsg()`. It contains all the details on the external message or status to be read or posted.

```
struct jobExternalMsgReq {
    int         options;    Indicated which operation to be performed
#define EXT_MSG_POST 0x01   Post external message
#define EXT_ATTA_POST 0x02  Post external data file
#define EXT_MSG_READ 0x04   Read external message
#define EXT_ATTA_READ 0x08  Read external data file
#define EXT_MSG_REPLAY 0x10 Replay external message
    LS_LONG_INT jobId;      Message of the job to be posted/read
    char        *jobName;   Name of the job if jobId is undefined (<=0)
    int         msgIdx;     Index in the list
    char        *desc;      Text description of the message
    int         userId;     Author of the message
    long        dataSize;   Size of the data file
    time_t      postTime;   Message sending time
};
```

The struct jobExternalMsgReply holds information on external message/status requested by the user. It is defined in `lsbatch.h` as follows:

```
struct jobExternalMsgReply {
    LS_LONG_INT jobId;       Message of the job to be read
    int         msgIdx;      Index in the message list
    char        *desc;       Text description of the message
    int         userId;      Author of the message
    long        dataSize;    Size of the data file
    time_t      postTime;    Message sending time
    int         dataStatus;  Status of the attached data
#define EXT_DATA_UNKNOWN 0   Data transferring of the message is processing
#define EXT_DATA_NOEXIST 1   Message without data attached
#define EXT_DATA_AVAIL 2     Data of the message is available
#define EXT_DATA_UNAVAIL 3   Data of the message is corrupt
};
```

User and host related functions

User information

Use `lsb.users` to:

- Configure user groups, hierarchical fairshare for users and user groups, and job slot limits for users and user groups.
- Configure account mappings in a MultiCluster environment.

LSBLIB provides the function `lsb_userinfo()` for getting information on LSF user and user groups.

```
struct userInfoEnt *lsb_userinfo(users, numUsers)
    char **users;             User names
    int  *numUsers;           Number of user names
```

To get information about all users, set *numUsers = 0; *numUsers is updated to the actual number of users when `lsb_userinfo()` returns. To get information on the invoker, set users = NULL and *numUsers = 1.

The function returns an array of userInfoEnt structure containing user information. The structure is defined in `lsbatch.h` as followed:

```
struct userInfoEnt {
    char  *user;               Name of the user or user group
    float procJobLimit;        Max number of started jobs on each processor
    int   maxJobs;             Max number of started or running jobs allowed
    int   numStartJobs;        Number of started jobs of the user/group
    int   numJobs;             Number of jobs the user/group submitted
    int   numPEND;             Number of pending jobs of the user/group
    int   numRUN;              Number of running jobs of the user/group
    int   numSSUSP;            Number of system-suspended jobs
    int   numUSUSP;            Number of user-suspended jobs
    int   numRESERVE;          Number of job slots reserved for pending jobs
};
```

`lsb_userinfo()` gets:

- The maximum number of job slots that a user can use simultaneously on any host
- The maximum number of job slots that a user can use simultaneously in the whole local LSF cluster
- The current number of job slots used by running and suspended jobs
- The current number of job slots reserved for pending jobs

The maximum number of job slots are defined in the `lsb.users` LSF configuration file. The reserved user name default, also defined in `lsb.users`, matches users not already listed in `lsb.users` who have no jobs started in the system.

On success, returns an array of userInfoEnt structures and sets *numUsers to the number of userInfoEnt structures returned. The next call writes over the returned array.

On failure, `lsb_userinfo()` returns `NULL` and sets `lsberrno` to indicate the error. If `lsberrno` is `LSBE_BAD_USER`, (*users)[*numUsers] is not a user known to the LSF system. Otherwise, if *numUsers is less than its original value, *numUsers is the actual number of users found.

Information in host group or user group

`lsb_hostgrpinfo()` and `lsb_usergrpinfo()` get membership of LSF host or user groups.

```
struct groupInfoEnt *lsb_hostgrpinfo (groups,
numGroups,                                       options)
struct groupInfoEnt *lsb_usergrpinfo (groups,
numGroups,                                      options)
    char  **groups;            Array of group names
    int   *numGroups;          Number of group names
    int   options;
struct groupInfoEnt {
    char   *group;            Group name
    char   *memberList;       ASCII list of member names
    int    numUserShares;     Number of users with shares
    struct userShares *userShares; User shares representation
};
struct userShares {
    char   *user;             User name
    int    shares;            Number of shares assigned to the user
};
    options                  The bitwise inclusive OR of some of the
                              following flags:
```

USER_GRP

Get the information of user group.

HOST_GRP

Get the information of host.

GRP_RECURSIVE

Expand the group membership recursively. That is, if a member of a group is itself a group, give the names of its members recursively, rather than its name, which is the default.

GRP_ALL

Get membership of all groups.

GRP_SHARES

Display the information in the long format.

`lsb_hostgrpinfo()` gets LSF host group membership, `lsb_usergrpinfo()` gets LSF user group membership.

`lsb.users(5)` and `lsb.hosts(5)` define LSF user and host groups, respectively.

On success, `lsb_hostgrpinfo()` and `lsb_usergrpinfo()` return an array of groupInfoEnt structures which hold the group name and the list of names of its members. If a member of a group is itself a group (i.e., a subgroup), then a '/' is appended to the name to indicate this. *numGroups is the number of groupInfoEnt structures returned.

On failure, `lsb_hostgrpinfo()` and `lsb_usergrpinfo()` returns `NULL` and sets `lsberrno` to indicate the error. If `lsberrno` is `LSBE_BAD_GROUP`, (*groups)[*numGroups] is not a group known to the LSF system. Otherwise, if *numGroups is less than its original value, *numGroups is the actual number of groups found.

Host partition in fairshare scheduling

To configure host partition fairshare, define a host partition in `lsb.hosts`. `lsb_hostpartinfo()` to gets the information on defined host partitions.

```
struct hostPartInfoEnt *lsb_hostpartinfo
(hostParts,                                         numHostParts)
    char **hostParts;         Host partition names
    int  *numHostParts;       Number of host partition names
```

To get information on all host partitions, set hostParts to NULL; *numHostParts is the actual number of host partitions when this `lsb_hostpartinfo()` returns.

The next call reuses the storage for the array of hostPartInfoEnt structures.

`lsb_hostpartinfo()` returns a struct hostPartInfoEnt describing the host partitions:

```
struct hostPartInfoEnt {
    char hostPart[MAX_LSB_NAME_LEN]; Name of the host partition
    char *hostList;                  Names of hosts in the partition
    int  numUsers;                   Number of users sharing the partition
    struct hostPartUserInfo *users; Description of user in the partition
};
```

The string variable hostList contains the names of the host in the partition and each of the names has a foward slash character (`/`) appended. (See `lsb_groupinfo(3)`.)

The struct hostPartUserInfo holds information on a specific user in the host partition.

```
struct hostPartUserInfo {
    char user[MAX_LSB_NAME_LEN]; User Name
    int   shares;                Number of shares assigned to the user
    float priority;              Priority of user to use the host partition
    int   numStartJobs;          Number of started jobs on host partition
    float histCpuTime;           Normalized CPU time of finished jobs
    int   numReserveJobs;        Number of reserved job slots for pending
                                 jobs
    int   runTime;               Time unfinished jobs spend in RUN state
};
```

For priority, the bigger values represent higher priorities. Jobs belonging to the user or user group with the highest priority are considered first for dispatch when resources in the host partition are being contended for. In general, a user or user group with more shares, fewer numStartJobs and less histCpuTime has higher priority.

On success, returns an array of hostPartInfoEnt structures which hold information on the host partitions, and sets *numHostParts to the number of hostPartInfoEnt structures.

On failure, `lsb_hostpartinfo()` returns NULL and sets lsberrno to indicate the error. If `lsberrno` is `LSBE_BAD_HPART`, (*hostParts)[*numHostParts] is not a host partition known to the LSF system. Otherwise, if *numHostParts is less than its original value, *numHostParts is the actual number of host partitions found.

Control hosts and daemons

The user can control the hosts and daemons through `lsb_hostcontrol()` and `lsb_reconfig()`.

`lsb_hostcontrol()` opens or closes a host and restarts or shutdowns the slave batch daemon.

```
int  lsb_hostcontrol (struct hostCtrlReq *);
struct hostCtrlReq {    char  *host;            Host to be controlled
int    opCode;           Option for host control    char
*message;          Message attached by the admin
};
```

If host is NULL, the local host is assumed.

`lsbatch.h` defines the opCode parameter containing the following control selection flags:

HOST_CLOSE

Closes the host so that no jobs can dispatched to it.

HOST_OPEN

Opens the host to accept jobs.

HOST_REBOOT

Restart the `sbatchd` on the host. The sbatchd will receive a request from the `mbatchd` and re-execute itself. This permits the `sbatchd` binary to be updated. This operation will fail if no sbatchd is running on the specified host.

HOST_SHUTDOWN

The `sbatchd` on the host will exit.

HOST_CLOSE_REMOTE

MultiCluster — Closes a leased host on the submission cluster

In order to use updated batch LSF configuration files, the user can use `lsb_reconfig()` to restart the master batch daemon, `mbatchd`.

```
int  lsb_reconfig (struct mbdCtrlReq *);
struct mbdCtrlReq {    int    opCode;          Options for configuration
char   *name;         Reserved for future use    char  *message;       Message
attached by the admin };
```

The parameter opCode is defined in `lsbatch.h` and should be one of the following:

MBD_RESTART

Restarts a new `mbatchd`

MBD_RECONFIG

Reread the configuration files

MBD_CKCONFIG

Check validity of the `mbatchd` configuration files

`lsb_reconfig()` provides the following functionality to:

- Dynamically reconfigure an LSF batch system to pick up new configuration parameters
- Change to the job queue setup since system startup or the last reconfiguration
- Restart a new master batch daemon
- Check the validity of the configuration files.

On success, both `lsb_hostcontrol()` and `lsb_reconfig()`. On failure, they return -1 and set `lsberrno` to indicate the error.

Index