Linux® Resource Administration Guide

007–4413–004

CONTRIBUTORS

Written by Terry Schultz

Illustrated by Chris Wengelski

Production by Karen Jacobson

Engineering contributions by Jeremy Brown, Marlys Kohnke, Paul Jackson, John Hesterberg, Robin Holt, Kevin McMahon, Troy Miller, Dennis Parker, Sam Watters, and Todd Wyman

New Features in This Manual

This rewrite of the *Linux Resource Administration Guide* supports the 2.4 release of the SGI ProPack for Linux operating system.

New Features Documented

Added information about the bootcpuset facility in "Bootcpuset" on page 54.

Major Documentation Changes

Removed Chapter 2 on Comprehensive System Accounting (CSA).

Record of Revision

Version	Description
001	February 2003 Original publication.
002	June 2003 Updated to support the SGI ProPack for Linux 2.2 release
003	October 2003 Updated to support the SGI ProPack for Linux 2.3 release.
004	February 2004 Updated to support the SGI ProPack for Linux 2.4 release.

Contents

Figures

Tables

About This Guide

This guide is a reference document for people who manage the operation of SGI computer systems running the Linux operating system. It contains information needed in the administration of various system resource management features.

This manual contains the following chapters:

- Chapter 1, "Linux Kernel Jobs" on page 1
- Chapter 2, "Array Services" on page 5
- Chapter 3, "CPU Memory Sets and Scheduling" on page 37
- Chapter 4, "Cpuset System" on page 51
- Chapter 5, "NUMA Tools" on page 67
- Appendix A, "Application Programming Interface for the Cpuset System" on page 69

Related Publications

For a list of Array Services man pages, see "Using Array Services Commands" on page 12.

Obtaining Publications

You can obtain SGI documentation in the following ways:

- See the SGI Technical Publications Library at: http://docs.sgi.com. Various formats are available. This library contains the most recent and most comprehensive set of online books, release notes, man pages, and other information.
- SGI ProPack for Linux documentation, and all other documentation included in the RPMs on the distribution CDs, can be found on the CD titled "SGI ProPack V.2.4 for Linux - Documentation CD." To access the information on the documentation CD, open the `index.html` file with a web browser. Because this online file can be updated later in the release cycle than this document, you should check it for the latest information. After installation, all SGI ProPack for

Linux documentation (including `README.SGI`) is in the `/usr/share/doc/sgi-propack-2.4` directory.

- You can view man pages by typing man *title* on a command line.

Conventions

The following conventions are used throughout this document:

Convention	Meaning
`command`	This fixed-space font denotes literal items such as commands, files, routines, path names, signals, messages, and programming language structures.
variable	Italic typeface denotes variable entries and words or concepts being defined.
`user input`	This bold, fixed-space font denotes literal items that the user enters in interactive sessions. (Output is shown in nonbold, fixed-space font.)
[ ]	Brackets enclose optional portions of a command or directive line.
...	Ellipses indicate that a preceding element can be repeated.

Reader Comments

If you have comments about the technical accuracy, content, or organization of this publication, contact SGI. Be sure to include the title and document number of the publication with your comments. (Online, the document number is located in the front matter of the publication. In printed publications, the document number is located at the bottom of each page.)

You can contact SGI in any of the following ways:

- Send e-mail to the following address:

  techpubs@sgi.com

- Use the Feedback option on the Technical Publications Library Web page:

http://docs.sgi.com

- Contact your customer service representative and ask that an incident be filed in the SGI incident tracking system.
- Send mail to the following address:

  Technical Publications
  SGI
  1500 Crittenden Lane, M/S 535
  Mountain View, California 94043–1351

SGI values your comments and will respond to them promptly.

Linux Kernel Jobs

This chapter describes Linux kernel jobs and contains the following sections:

Overview

Work on a machine is submitted in a variety of ways, such as an interactive login, a submission from a workload management system, a `cron` job, or a remote access such as `rsh`, `rcp`, or array services. Each of these points of entry creates an original shell process and multiple processes flow from that original point of entry. The Linux kernel job, used by the Comprehensive System Accounting (CSA) software, provides a means to measure the resource usage of all the processes resulting from a point of entry. A job is a group of related processes all descended from a point-of- entry process and identified by a unique job ID. A job can contain multiple process groups, sessions, or array sessions and all processes in one of these subgroups are always contained within one job. Figure 1-1 on page 2, shows the point-of-entry processes that initiate the creation of jobs.

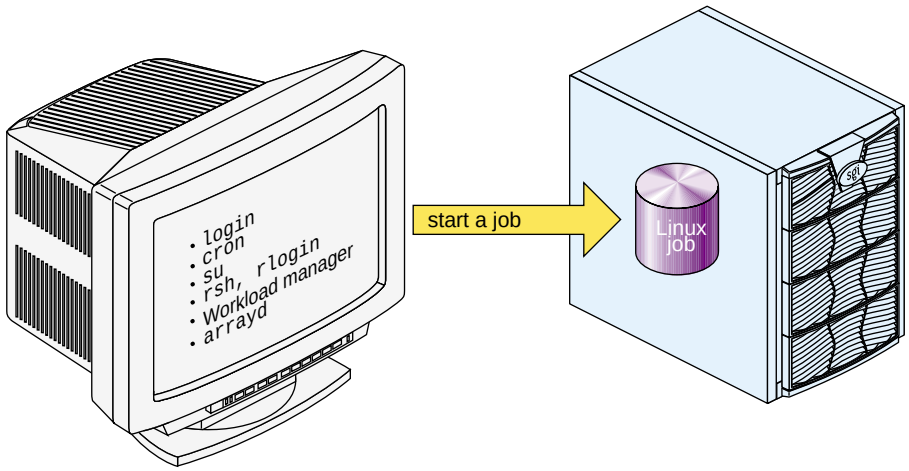

Figure 1-1 Point-of-Entry Processes

A Linux job has the following characteristics:

- A job is an inescapable container. A process cannot leave the job nor can a new process be created outside the job without explicit action, that is, a system call with root privilege.
- Each new process inherits the job ID from its parent process.
- All point-of-entry processes (job initiators) create a new job.
- The job initiator performs authentication and security checks.
- Job initiation on Linux is performed via a Pluggable Authentication Module (PAM) session module.

  Note: PAMs are a suite of shared libraries that enable the local system administrator to choose how applications authenticate users. For more information on PAM, see the *Linux Configuration and Operations Guide*.

- Not all processes on a system need to be members of a job.

The process-control initialization process (`init`(8)) and startup scripts called by `init` are not part of a job and have a job ID of zero.

Note: The existing command `jobs`(1) applies to shell "jobs" and it is not related to the Linux kernel module jobs. The `at`(1), `atd`(8), `atq`(1), `batch`(1), `atrun`(8), and `atrm`(1) man pages refer to shell scripts as a job.

Installing and Configuring Linux Kernel Jobs

Linux kernel jobs are part of the kernel on your SGI ProPack for Linux system. To configure jobs for services, such as Comprehensive System Accounting (CSA), perform the following steps:

1. Change to the directory where the PAM configuration files reside by entering the following:

   ```
   cd /etc/pam.d
   ```

2. To enable job creation for all session services add an entry to the `/etc/pam.d/system-auth` file.

   If you want to enable jobs only for certain PAM services you can update individual configuration files. This example shows the `login` configuration file being changed. You customize PAM services by adding the `session` line to PAM entry points that will create jobs on your system, for example, `login`, `rlogin`, `rsh`, and `su`.

   To enable job creation for `login` users by adding this entry to the `login` configuration file:

   ```
   session   required    /lib/security/pam_job.so
   ```

3. To configure jobs to be started automatically during system startup, use the `chkconfig`(8) command as follows:

   ```
   chkconfig --add job
   ```

4. To stop jobs from being started automatically during system startup, use the `chkconfig`(8) command as follows:

   ```
   chkconfig --del job
   ```

Array Services

Array Services includes administrator commands, libraries, daemons, and kernel extensions that support the execution of programs across an array.

A central concept in Array Services is the array session handle (ASH), a number that is used to logically group related processes that may be distributed across multiple systems. The ASH creates a global process namespace across the Array, facilitating accounting and administration

Array Services also provides an array configuration database, listing the nodes comprising an array. Array inventory inquiry functions provide a centralized, canonical view of the configuration of each node. Other array utilities let the administrator query and manipulate distributed array applications.

This chapter covers the follow topics:

Array Services Package

The Array Services package comprises the following primary components:

array daemon	Allocates ASH values and maintain information about node configuration and the relation of process IDs to ASHs. Array daemons reside on each node and work in cooperation.
array configuration database	Describes the array configuration used by array daemons and user programs. One copy at each node.
`ainfo` command	Lets the user or administrator query the Array configuration database and information about ASH values and processes.
`array` command	Executes a specified command on one or more nodes. Commands are predefined by the administrator in the configuration database.
`arshell` command	Starts a command remotely on a different node using the current ASH value.
`aview` command	Displays a multiwindow, graphical display of each node's status. (Not currently available)

The use of the `ainfo`, `array`, `arshell`, and `aview` commands is covered in "Using an Array" on page 8.

Installing and Configuring Array Services

To use the Array Services package on Linux, you must have an Array Services enabled kernel. This is done with the `arsess` kernel module, which is provided with SGI's Linux Base Software. If the module is installed correctly, the `init` script provided with the Array Services rpm will load the module when starting up the `arrayd` daemon.

1. An account must exist on all hosts in the array for the purposes of running certain Array Services commands. This is controlled by the `/usr/lib/array/arrayd.conf` configuration file. The default is to use the user account "guest" since this is typically found on UNIX machines. The account name can be changed in `arrayd.conf`. For more information, see the `arrayd.conf`(8) man page.

If necessary, add the specified user account or "guest" by default, to all machines in the array.

2. Add the following entry to `/etc/services` file for `arrayd` service and port. The default port number is 5434 and is specified in the `arrayd.conf` configuration file.

```
sgi-arrayd   5434/tcp    # SGI Array Services daemon
```

3. If necessary, modify the default authentication configuration. The default authentication is `AUTHENTICATION NOREMOTE`, which does not allow access from remote hosts. The authentication model is specified in the `/usr/lib/array/arrayd.auth` configuration file.

4. To configure Array Services on across system reboots using the `chkconfig`(8) utility, perform the following:

```
chkconfig --add array
```

5. For information on configuring Array Services, see the following:

   - "About Array Configuration" on page 24
   - "Configuring Arrays and Machines" on page 29
   - "Configuring Authentication Codes" on page 30
   - "Configuring Array Commands" on page 31

6. To turn on Array Services, perform the following:

```
/etc/rc.d/init.d/array start
```

   This step will be done automatically for subsequent system reboots when Array Services is configured on via the `chkconfig`(8) utility.

The following steps are required to disable Array Services:

1. To turn off Array Services, perform the following:

```
/etc/rc.d/init.d/array stop
```

2. To stop Array Services from initiating after a system reboot, use the `chkconfig`(8) command:

```
chkconfig --del array
```

Using an Array

An Array system is an aggregation of nodes, which are servers bound together with a high-speed network and Array Services 3.5 software. Array users have the advantage of greater performance and additional services. Array users access the system with familiar commands for job control, login and password management, and remote execution.

Array Services 3.5 augments conventional facilities with additional services for array users and for array administrators. The extensions include support for global session management, array configuration management, batch processing, message passing, system administration, and performance visualization.

This section introduces the extensions for Array use, with pointers to more detailed information. The main topics are as follows:

- "Using an Array System" on page 8, summarizes what a user needs to know and the main facilities a user has available.
- "Managing Local Processes" on page 11, reviews the conventional tools for listing and controlling processes within one node.
- "Using Array Services Commands" on page 12, describes the common concepts, options, and environment variables used by the Array Services commands.
- "Interrogating the Array" on page 16, summarizes how to use Array Services commands to learn about the Array and its workload, with examples.
- "Summary of Common Command Options" on page 14
- "Managing Distributed Processes" on page 19, summarizes how to use Array Services commands to list and control processes in multiple nodes.

Using an Array System

The array system allows you to run distributed sessions on multiple nodes of an array. You can access the Array from either:

- A workstation
- An X terminal
- An ASCII terminal

In each case, you log in to one node of the Array in the way you would log in to any remote UNIX host. From a workstation or an X terminal you can of course open more than one terminal window and log into more than one node.

Finding Basic Usage Information

In order to use an Array, you need the following items of information:

- The name of the Array.

  You use this *arrayname* in Array Services commands.

- The login name and password you will use on the Array.

  You use these when logging in to the Array to use it.

- The hostnames of the array nodes.

  Typically these names follow a simple pattern, often *arrayname1*, *arrayname2*, and so on.

- Any special resource-distribution or accounting rules that may apply to you or your group under a job scheduling system.

You can learn the hostnames of the array nodes if you know the array name, using the `ainfo` command as follows:

`ainfo -a` *arrayname* **`machines`**

Logging In to an Array

Each node in an Array has an associated hostname and IP network address. Typically, you use an Array by logging in to one node directly, or by logging in remotely from another host (such as the Array console or a networked workstation). For example, from a workstation on the same network, this command would log you in to the node named `hydra6` as follows:

```
rlogin hydra6
```

For details of the `rlogin` command, see the `rlogin`(1) man page.

The system administrators of your array may choose to disallow direct node logins in order to schedule array resources. If your site is configured to disallow direct node logins, your administrators will be able to tell you how you are expected to submit

work to the array–perhaps through remote execution software or batch queueing facilities.

Invoking a Program

Once you have access to an array, you can invoke programs of several classes:

- Ordinary (sequential) applications
- Parallel shared-memory applications within a node
- Parallel message-passing applications within a node
- Parallel message-passing applications distributed over multiple nodes (and possibly other servers on the same network running Array Services 3.5

If you are allowed to do so, you can invoke programs explicitly from a logged-in shell command line; or you may use remote execution or a batch queueing system.

Programs that are X Windows clients must be started from an X server, either an X Terminal or a workstation running X Windows.

Some application classes may require input in the form of command line options, environment variables, or support files upon execution. For example:

- X client applications need the `DISPLAY` environment variable set to specify the X server (workstation or X-terminal) where their windows will display.
- A multithreaded program may require environment variables to be set describing the number of threads.

  For example, C and Fortran programs that use parallel processing directives test the `MP_SET_NUMTHREADS` variable.
- Message Passing Interface (MPI) and Parallel Virtual Machine (PVM) message-passing programs may require support files to describe how many tasks to invoke on specified nodes.

Some information sources on program invocation are listed in Table 2-1 on page 11.

Table 2-1 Information Sources for Invoking a Program

Topic	Man Page
Remote login	`rlogin`(1)
Setting environment variables	`environ`(5), `env`(1)

Managing Local Processes

Each UNIX process has a *process identifier* (PID), a number that identifies that process within the node where it runs. It is important to realize that a PID is local to the node; so it is possible to have processes in different nodes using the same PID numbers.

Within a node, processes can be logically grouped in *process groups*. A process group is composed of a parent process together with all the processes that it creates. Each process group has a *process group identifier* (PGID). Like a PID, a PGID is defined locally to that node, and there is no guarantee of uniqueness across the Array.

Monitoring Local Processes and System Usage

You query the status of processes using the system command `ps`. To generate a full list of all processes on a local system, use a command such as the following:

```
ps -elfj
```

You can monitor the activity of processes using the command `top` (an ASCII display in a terminal window).

Scheduling and Killing Local Processes

You can schedule commands to run at specific times using the `at` command. You can kill or stop processes using the `kill` command. To destroy the process with PID 13032, use a command such as the following:

```
kill -KILL 13032
```

Summary of Local Process Management Commands

Table 2-2 on page 12, summarizes information about local process management.

Table 2-2 Information Sources: Local Process Management standard

Topic	Man Page
Process ID and process group	`intro`(2)
Listing and monitoring processes	`ps`(1), `top`(1)
Running programs at low priority	`nice`(1), `batch`(1)
Running programs at a scheduled time	`at`(1)
Terminating a process	`kill`(1)

Using Array Services Commands

When an application starts processes on more than one node, the PID and PGID are no longer adequate to manage the application. The commands of Array Services 3.5 give you the ability to view the entire array, and to control the processes of multinode programs.

Note: You can use Array Services commands from any workstation connected to an array system. You don't have to be logged in to an array node.

The following commands are common to Array Services operations as shown in Table 2-3 on page 13.

Table 2-3 Common Array Services Commands

Topic	Man Page
Array Services Overview	`array_services`(5)
`ainfo` command	`ainfo`(1)
`array` command	Use `array`(1); configuration: `arrayd.conf`(4)
`arshell` command	`arshell`(1)
`newsess` command	`newsess` (1)

About Array Sessions

Array Services is composed of a daemon–a background process that is started at boot time in every node–and a set of commands such as `ainfo`(1). The commands call on the daemon process in each node to get the information they need.

One concept that is basic to Array Services is the *array session*, which is a term for all the processes of one application, wherever they may execute. Normally, your login shell, with the programs you start from it, constitutes an array session. A batch job is an array session; and you can create a new shell with a new array session identity.

Each session is identified by an *array session handle* (ASH), a number that identifies any process that is part of that session. You use the ASH to query and to control all the processes of a program, even when they are running in different nodes.

About Names of Arrays and Nodes

Each node is server, and as such has a hostname. The hostname of a node is returned by the `hostname`(1) command executed in that node as follows:

```
% hostname
tokyo
```

The command is simple and documented in the `hostname`(1) man page. The more complicated issues of hostname syntax, and of how hostnames are resolved to hardware addresses are covered in `hostname`(5).

An Array system as a whole has a name too. In most installations there is only a single Array, and you never need to specify which Array you mean. However, it is possible to have multiple Arrays available on a network, and you can direct Array Services commands to a specific Array.

About Authentication Keys

It is possible for the Array administrator to establish an authentication code, which is a 64-bit number, for all or some of the nodes in an array (see "Configuring Authentication Codes" on page 58). When this is done, each use of an Array Services command must specify the appropriate authentication key, as a command option, for the nodes it uses. Your system administrator will tell you if this is necessary.

Summary of Common Command Options

The following Array Services commands have a consistent set of command options: `ainfo`(1), `array`(1), `arshell`(1), and `aview`(1) (`aview`(1) is not currently available). Table 2-4 is a summary of these options. Not all options are valid with all commands; and each command has unique options besides those shown. The default values of some options are set by environment variables listed in the next topic.

Table 2-4 Array Services Command Option Summary

Option	Used In	Description
`-a` *array*	`ainfo`, `array`, `aview`	Specify a particular Array when more than one is accessible.
`-D`	`ainfo`, `array`, `arshell`, `aview`	Send commands to other nodes directly, rather than through `array` daemon.

Option	Used In	Description
`-F`	`ainfo`, `array`, `arshell`, `aview`	Forward commands to other nodes through the `array` daemon.
`-Kl` *number*	`ainfo`, `array`, `aview`	Authentication key (a 64-bit number) for the local node.
`-Kr` *number*	`ainfo`, `array`, `aview`	Authentication key (a 64-bit number) for the remote node.
`-l` (letter ell)	`ainfo`, `array`	Execute in context of the destination node, not necessarily the current node.
`-l` *port*	`ainfo`, `array`, `arshell`, `aview`	Nonstandard port number of array daemon.
`-s` *hostname*	`ainfo`, `array`, `aview`	Specify a destination node.

Specifying a Single Node

The `-l` and `-s` options work together. The `-l` (letter ell for "local") option restricts the scope of a command to the node where the command is executed. By default, that is the node where the command is entered. When `-l` is not used, the scope of a query command is all nodes of the array. The `-s` (server, or node name) option directs the command to be executed on a specified node of the array. These options work together in query commands as follows:

- To interrogate all nodes as seen by the local node, use neither option.
- To interrogate only the local node, use only `-l`.
- To interrogate all nodes as seen by a specified node, use only `-s`.
- To interrogate only a particular node, use both `-s` and `-l`.

Common Environment Variables

The Array Services commands depend on environment variables to define default values for the less-common command options. These variables are summarized in Table 2-5.

Table 2-5 Array Services Environment Variables

Variable Name	Use	Default When Undefined
ARRAYD_FORWARD	When defined with a string starting with the letter *y*, all commands default to forwarding through the array daemon (option `-F`).	Commands default to direct communication (option `-D`).
ARRAYD_PORT	The port (socket) number monitored by the array daemon on the destination node.	The standard number of 5434, or the number given with option `-p`.
ARRAYD_LOCALKEY	Authentication key for the local node (option `-Kl`).	No authentication unless `-Kl` option is used.
ARRAYD_REMOTEKEY	Authentication key for the destination node (option `-Kr`).	No authentication unless `-Kr` option is used.
ARRAYD	The destination node, when not specified by the `-s` option.	The local node, or the node given with `-s`.

Interrogating the Array

Any user of an Array system can use Array Services commands to check the hardware components and the software workload of the Array. The commands needed are `ainfo`, `array`, and `aview`.

Learning Array Names

If your network includes more than one Array system, you can use `ainfo arrays` at one array node to list all the Array names that are configured, as in the following example.

```
homegrown% ainfo arrays
Arrays known to array services daemon
ARRAY DevArray
    IDENT 0x3381
ARRAY BigDevArray
    IDENT 0x7456
ARRAY test
    IDENT 0x655e
```

Array names are configured into the array database by the administrator. Different Arrays might know different sets of other Array names.

Learning Node Names

You can use `ainfo machines` to learn the names and some features of all nodes in the current Array, as in the following example.

```
homegrown 175% ainfo -b machines
machine homegrown homegrown 5434 192.48.165.36 0
machine disarray disarray 5434 192.48.165.62 0
machine datarray datarray 5434 192.48.165.64 0
machine tokyo tokyo 5434 150.166.39.39 0
```

In this example, the `-b` option of `ainfo` is used to get a concise display.

Learning Node Features

You can use `ainfo nodeinfo` to request detailed information about one or all nodes in the array. To get information about the local node, use `ainfo -l nodeinfo`. However, to get information about only a particular other node, for example node `tokyo`, use `-l` and `-s`, as in the following example. (The example has been edited for brevity.)

```
homegrown 181% ainfo -s tokyo -l nodeinfo
Node information for server on machine "tokyo"
MACHINE tokyo
    VERSION  1.2
    8 PROCESSOR BOARDS
        BOARD: TYPE 15   SPEED 190
            CPU:   TYPE 9   REVISION 2.4
            FPU:   TYPE 9   REVISION 0.0
```

```
...
    16 IP INTERFACES  HOSTNAME tokyo   HOSTID 0xc01a5035
        DEVICE  et0    NETWORK    150.166.39.0    ADDRESS   150.166.39.39  UP
        DEVICE atm0    NETWORK 255.255.255.255    ADDRESS         0.0.0.0  UP
        DEVICE atm1    NETWORK 255.255.255.255    ADDRESS         0.0.0.0  UP
...
    0 GRAPHICS INTERFACES
    MEMORY
        512 MB MAIN MEMORY
        INTERLEAVE 4
```

If the `-l` option is omitted, the destination node will return information about every node that it knows.

Learning User Names and Workload

The system commands `who`(1), `top`(1), and `uptime`(1) are commonly used to get information about users and workload on one server. The `array`(1) command offers Array-wide equivalents to these commands.

Learning User Names

To get the names of all users logged in to the whole array, use `array who`. To learn the names of users logged in to a particular node, for example `tokyo`, use `-l` and `-s`, as in the following example. (The example has been edited for brevity and security.)

```
homegrown 180% array -s tokyo -l who
joecd    tokyo        frummage.eng.sgi -tcsh
joecd    tokyo        frummage.eng.sgi -tcsh
benf     tokyo        einstein.ued.sgi. /bin/tcsh
yohn     tokyo        rayleigh.eng.sg vi +153 fs/procfs/prd
...
```

Learning Workload

Two variants of the `array` command return workload information. The array-wide equivalent of `uptime` is `array uptime`, as follows:

```
homegrown 181% array uptime
   homegrown:  up 1 day,  7:40,  26 users,  load average: 7.21, 6.35, 4.72
    disarray:  up  2:53,  0 user,  load average: 0.00, 0.00, 0.00
```

```
    datarray:  up  5:34,  1 user,  load average: 0.00, 0.00, 0.00
       tokyo:  up 7 days,  9:11,  17 users,  load average: 0.15, 0.31, 0.29
homegrown 182% array -l -s tokyo uptime
       tokyo:  up 7 days,  9:11,  17 users,  load average: 0.12, 0.30, 0.28
```

The command `array top` lists the processes that are currently using the most CPU time, with their ASH values, as in the following example.

```
homegrown 183% array top
        ASH        Host           PID User       %CPU Command
-------------------------------------------------------------
0x1111ffff00000000 homegrown        5 root       1.20 vfs_sync
0x1111ffff000001e9 homegrown     1327 guest      1.19 atop
0x1111ffff000001e9 tokyo        19816 guest      0.73 atop
0x1111ffff000001e9 disarray      1106 guest      0.47 atop
0x1111ffff000001e9 datarray      1423 guest      0.42 atop
0x1111ffff00000000 homegrown       20 root       0.41 ShareII
0x1111ffff000000c0 homegrown    29683 kchang     0.37 ld
0x1111ffff0000001e homegrown     1324 root       0.17 arrayd
0x1111ffff00000000 homegrown      229 root       0.14 routed
0x1111ffff00000000 homegrown       19 root       0.09 pdflush
0x1111ffff000001e9 disarray      1105 guest      0.02 atopm
```

The `-l` and `-s` options can be used to select data about a single node, as usual.

Managing Distributed Processes

Using commands from Array Services 3.5, you can create and manage processes that are distributed across multiple nodes of the Array system.

About Array Session Handles (ASH)

In an Array system you can start a program with processes that are in more than one node. In order to name such collections of processes, Array Services 3.5 software assigns each process to an *array session handle* (ASH).

An ASH is a number that is unique across the entire array (unlike a PID or PGID). An ASH is the same for every process that is part of a single array session—no matter which node the process runs in. You display and use ASH values with Array Services

commands. Each time you log in to an Array node, your shell is given an ASH, which is used by all the processes you start from that shell.

The command `ainfo ash` returns the ASH of the current process on the local node, which is simply the ASH of the `ainfo` command itself.

```
homegrown 178% ainfo ash
Array session handle of process 10068: 0x1111ffff000002c1
homegrown 179% ainfo ash
Array session handle of process 10069: 0x1111ffff000002c1
```

In the preceding example, each instance of the `ainfo` command was a new process: first PID 10068, then PID 10069. However, the ASH is the same in both cases. This illustrates a very important rule: **every process inherits its parent's ASH**. In this case, each instance of `array` was forked by the command shell, and the ASH value shown is that of the shell, inherited by the child process.

You can create a new global ASH with the command `ainfo newash`, as follows:

```
homegrown 175% ainfo newash
Allocating new global ASH
0x11110000308b2f7c
```

This feature has little use at present. There is no existing command that can change its ASH, so you cannot assign the new ASH to another command. It is possible to write a program that takes an ASH from a command-line option and uses the Array Services function `setash()` to change to that ASH (however such a program must be privileged). No such program is distributed with Array Services 3.5.

Listing Processes and ASH Values

The command `array ps` returns a summary of all processes running on all nodes in an array. The display shows the ASH, the node, the PID, the associated username, the accumulated CPU time, and the command string.

To list all the processes on a particular node, use the `-l` and `-s` options. To list processes associated with a particular ASH, or a particular username, pipe the returned values through `grep`, as in the following example. (The display has been edited to save space.)

```
homegrown 182% array -l -s tokyo ps | fgrep wombat
0x261cffff0000054c        tokyo 19007   wombat    0:00 -csh
0x261cffff0000054a        tokyo 17940   wombat    0:00 csh -c (setenv...
```

```
0x261cffff0000054c        tokyo 18941   wombat    0:00 csh -c (setenv...
0x261cffff0000054a        tokyo 17957   wombat    0:44 xem -geometry 84x42
0x261cffff0000054a        tokyo 17938   wombat    0:00 rshd
0x261cffff0000054a        tokyo 18022   wombat    0:00 /bin/csh -i
0x261cffff0000054a        tokyo 17980   wombat    0:03 /usr/gnu/lib/ema...
0x261cffff0000054c        tokyo 18928   wombat    0:00 rshd
```

Controlling Processes

The `arshell` command lets you start an arbitrary program on a single other node. The `array` command gives you the ability to suspend, resume, or kill all processes associated with a specified ASH.

Using arshell

The `arshell` command is an Array Services extension of the familiar `rsh` command; it executes a single system command on a specified Array node. The difference from `rsh` is that the remote shell executes under the same ASH as the invoking shell (this is not true of simple `rsh`). The following example demonstrates the difference.

```
homegrown 179% ainfo ash
Array session handle of process 8506: 0x1111ffff00000425
homegrown 180% rsh guest@tokyo ainfo ash
Array session handle of process 13113: 0x261cffff0000145e
homegrown 181% arshell guest@tokyo ainfo ash
Array session handle of process 13119: 0x1111ffff00000425
```

You can use `arshell` to start a collection of unrelated programs in multiple nodes under a single ASH; then you can use the commands described under "Managing Session Processes" on page 23 to stop, resume, or kill them.

Both MPI and PVM use `arshell` to start up distributed processes.

Tip: The shell is a process under its own ASH. If you use the `array` command to stop or kill all processes started from a shell, you will stop or kill the shell also. In order to create a group of programs under a single ASH that can be killed safely, proceed as follows:

1. Within the new shell, start one or more programs using `arshell`.
2. Exit the nested shell.

Now you are back to the original shell. You know the ASH of all programs started from the nested shell. You can safely kill all jobs that have that ASH because the current shell is not affected.

About the Distributed Example

The programs launched with `arshell` are not coordinated (they could of course be written to communicate with each other, for example using sockets), and you must start each program individually.

The `array` command is designed to permit the simultaneous launch of programs on all nodes with a single command. However, `array` can only launch programs that have been configured into it, in the Array Services configuration file. (The creation and management of this file is discussed under "About Array Configuration" on page 24.)

In order to demonstrate process management in a simple way from the command line, the following command was inserted into the configuration file `/usr/lib/array/arrayd.conf`:

```
#
# Local commands
#
command spin                    # Do nothing on multiple machines
        invoke /usr/lib/array/spin
        user    %USER
        group   %GROUP
        options nowait
```

The invoked command, `/usr/lib/array/spin`, is a shell script that does nothing in a loop, as follows:

```
#!/bin/sh
# Go into a tight loop
```

```
#
interrupted() {
        echo "spin has been interrupted - goodbye"
        exit 0
}
trap interrupted 1 2
while [ ! -f /tmp/spin.stop ]; do
        sleep 5
done
echo "spin has been stopped - goodbye"
exit 1
```

With this preparation, the command `array spin` starts a process executing that script on every processor in the array. Alternatively, `array -l -s nodename spin` would start a process on one specific node.

Managing Session Processes

The following command sequence creates and then kills a `spin` process in every node. The first step creates a new session with its own ASH. This is so that later, `array kill` can be used without killing the interactive shell.

```
homegrown 175% ainfo ash
Array session handle of process 8912: 0x1111ffff0000032d
homegrown 175% ainfo ash
Array session handle of process 8941: 0x11110000308b2fa6
```

In the new session with ASH 0x11110000308b2fa6, the command array `spin` starts the `/usr/lib/array/spin` script on every node. In this test array, there were only two nodes on this day, `homegrown` and `tokyo`.

```
homegrown 176% array spin
```

After exiting back to the original shell, the command `array ps` is used to search for all processes that have the ASH 0x11110000308b2fa6.

```
homegrown 177% exit
homegrown 178% homegrown 177%
homegrown 177% ainfo ash
Array session handle of process 9257: 0x1111ffff0000032d
homegrown 179% array ps | fgrep 0x11110000308b2fa6
0x11110000308b2fa6  homegrown  9033   guest   0:00 /bin/sh /usr/lib/array/spin
0x11110000308b2fa6  homegrown  9618  guest  0:00 sleep 5
0x11110000308b2fa6        tokyo 26021  guest   0:00 /bin/sh /usr/lib/array/spin
0x11110000308b2fa6      tokyo 26072  guest  0:00 sleep 5
0x1111ffff0000032d  homegrown  9642  guest  0:00 fgrep 0x11110000308b2fa6
```

There are two processes related to the `spin` script on each node. The next command kills them all.

```
homegrown 180% array kill 0x11110000308b2fa6
homegrown 181% array ps | fgrep 0x11110000308b2fa6
0x1111ffff0000032d  homegrown 10030  guest  0:00 fgrep 0x11110000308b2fa6
```

The command `array suspend 0x11110000308b2fa6` would suspend the processes instead (however, it is hard to demonstrate that a `sleep` command has been suspended).

About Job Container IDs

Array systems have the capability to forward job IDs (JIDs) from the initiating host. All of the processes running in the ASH across one or more nodes in an array also belong to the same job. For a complete description of the job container and it usage, see Chapter 1, "Linux Kernel Jobs" on page 1.

When processes are running on the initiating host, they belong to the same job as the initiating process and operate under the limits established for that job. On remote nodes, a new job is created using the same JID as the initiating process. Job limits for a job on remote nodes use the `systune` defaults and are set using the `systune`(1M) command on the initiating host.

About Array Configuration

The system administrator has to initialize the Array configuration database, a file that is used by the Array Services daemon in executing almost every `ainfo` and `array` command. For details about array configuration, see the man pages cited in Table 2-6.

Table 2-6 Information Sources: Array Configuration

Topic	Man Page
Array Services overview	`array_services`(5)
Array Services user commands	`ainfo`(1) , `array`(1)
Array Services daemon overview	`arrayd`(1m)
Configuration file format	`arrayd.conf`(4) , `/usr/lib/array/arrayd.conf.template`
Configuration file validator	`ascheck`(1)
Array Services simple configurator	`arrayconfig`(1m)

About the Uses of the Configuration File

The configuration files are read by the Array Services daemon when it starts. Normally it is started in each node during the system startup. (You can also run the daemon from a command line in order to check the syntax of the configuration files.)

The configuration files contain data needed by `ainfo` and `array`:

- The names of Array systems, including the current Array but also any other Arrays on which a user could run an Array Services command (reported by `ainfo`).
- The names and types of the nodes in each named Array, especially the hostnames that would be used in an Array Services command (reported by `ainfo`).
- The authentication keys, if any, that must be used with Array Services commands (required as `-Kl` and `-Kr` command options, see "Summary of Common Command Options" on page 14).
- The commands that are valid with the `array` command.

About Configuration File Format and Contents

A configuration file is a readable text file. The file contains entries of the following four types, which are detailed in later topics.

Array definition	Describes this array and other known arrays, including array names and the node names and types.
Command definition	Specifies the usage and operation of a command that can be invoked through the `array` command.
Authentication	Specifies authentication numbers that must be used to access the Array.
Local option	Options that modify the operation of the other entries or `arrayd`.

Blank lines, white space, and comment lines beginning with "#" can be used freely for readability. Entries can be in any order in any of the files read by `arrayd`.

Besides punctuation, entries are formed with a keyword-based syntax. Keyword recognition is not case-sensitive; however keywords are shown in uppercase in this text and in the man page. The entries are primarily formed from keywords, numbers, and quoted strings, as detailed in the man page `arrayd.conf`(4).

Loading Configuration Data

The Array Services daemon, `arrayd`, can take one or more filenames as arguments. It reads them all, and treats them like logical continuations (in effect, it concatenates them). If no filenames are specified, it reads `/usr/lib/array/arrayd.conf` and `/usr/lib/array/arrayd.auth`. A different set of files, and any other `arrayd` command-line options, can be written into the file `/etc/config/arrayd.options`, which is read by the startup script that launches `arrayd` at boot time.

Since configuration data can be stored in two or more files, you can combine different strategies, for example:

- One file can have different access permissions than another. Typically, `/usr/lib/array/arrayd.conf` is world-readable and contains the available `array` commands, while `/usr/lib/array/arrayd.auth` is readable only by root and contains authentication codes.

- One node can have different configuration data than another. For example, certain commands might be defined only in certain nodes; or only the nodes used for interactive logins might know the names of all other nodes.
- You can use NFS-mounted configuration files. You could put a small configuration file on each machine to define the Array and authentication keys, but you could have a larger file defining `array` commands that is NFS-mounted from one node.

After you modify the configuration files, you can make `arrayd` reload them by killing the daemon and restarting it in each machine. The script `/etc/rc.d/init.d/array` supports this operation:

To kill daemon, execute this command:

```
/etc/rc.d/init.d/array stop
```

To kill and restart the daemon in one operation; peform the following command:

```
/etc/rc.d/init.d/array restart
```

Note: On Linux systems, the script path name is `/etc/rc.d/init.d/array`.

The Array Services daemon in any node knows only the information in the configuration files available in that node. This can be an advantage, in that you can limit the use of particular nodes; but it does require that you take pains to keep common information synchronized. (An automated way to do this is summarized under "Designing New Array Commands" on page 35.)

About Substitution Syntax

The man page `arrayd.conf`(4) details the syntax rules for forming entries in the configuration files. An important feature of this syntax is the use of several kinds of text substitution, by which variable text is substituted into entries when they are executed.

Most of the supported substitutions are used in command entries. These substitutions are performed dynamically, each time the `array` command invokes a subcommand. At that time, substitutions insert values that are unique to the invocation of that subcommand. For example, the value `%USER` inserts the user ID of the user who is invoking the `array` command. Such a substitution has no meaning except during execution of a command.

Substitutions in other configuration entries are performed only once, at the time the configuration file is read by `arrayd`. Only environment variable substitution makes sense in these entries. The environment variable values that are substituted are the values inherited by `arrayd` from the script that invokes it, which is `/etc/rc.d/init.d/array`.

Testing Configuration Changes

The configuration files contain many sections and options (detailed in the section that follow this one). The Array Services command `ascheck` performs a basic sanity check of all configuration files in the array.

After making a change, you can test an individual configuration file for correct syntax by executing `arrayd` as a command with the `-c` and `-f` options. For example, suppose you have just added a new command definition to `/usr/lib/array/arrayd.local`. You can check its syntax with the following command:

```
arrayd -c -f /usr/lib/array/arrayd.local
```

When testing new commands for correct operation, you need to see the warning and error messages produced by `arrayd` and processes that it may spawn. The `stderr` messages from a daemon are not normally visible. You can make them visible by the following procedure:

1. On one node, kill the daemon.
2. In one shell window on that node, start `arrayd` with the options `-n -v`. Instead of moving into the background, it remains attached to the shell terminal.

   Note: Although `arrayd` becomes functional in this mode, it does not refer to `/etc/config/arrayd.options`, so you need to specify explicitly all command-line options, such as the names of nonstandard configuration files.

3. From another shell window on the same or other nodes, issue `ainfo` and `array` commands to test the new configuration data. Diagnostic output appears in the `arrayd` shell window.
4. Terminate `arrayd` and restart it as a daemon (without `-n`).

During steps 1, 2, and 4, the test node may fail to respond to `ainfo` and `array` commands, so users should be warned that the Array is in test mode.

Configuring Arrays and Machines

Each ARRAY entry gives the name and composition of an Array system that users can access. At least one ARRAY must be defined at every node, the array in use.

Note: ARRAY is a keyword.

Specifying Arrayname and Machine Names

A simple example of an ARRAY definition is a follows:

```
array simple
        machine congo
        machine niger
        machine nile
```

The arrayname `simple` is the value the user must specify in the `-a` option (see "Summary of Common Command Options" on page 14). One arrayname should be specified in a DESTINATION ARRAY local option as the default array (reported by `ainfo dflt`). Local options are listed under "Configuring Local Options" on page 34.

It is recommended that you have at least one array called `me` that just contains the localhost. The default `arrayd.conf` file has the `me` array defined as the default destination array.

The MACHINE subentries of ARRAY define the node names that the user can specify with the `-s` option. These names are also reported by the command `ainfo machines`.

Specifying IP Addresses and Ports

The simple MACHINE subentries shown in the example are based on the assumption that the hostname is the same as the machine's name to Domain Name Services (DNS). If a machine's IP address cannot be obtained from the given hostname, you must provide a HOSTNAME subentry to specify either a completely qualified domain name or an IP address, as follows:

```
array simple
        machine congo
            hostname congo.engr.hitech.com
            port 8820
```

```
machine niger
    hostname niger.engr.hitech.com
machine nile
    hostname "198.206.32.85"
```

The preceding example also shows how the PORT subentry can be used to specify that `arrayd` in a particular machine uses a different socket number than the default 5434.

Specifying Additional Attributes

Under both ARRAY and MACHINE you can insert attributes, which are named string values. These attributes are not used by Array Services, but they are displayed by `ainfo` .Some examples of attributes would be as follows:

```
array simple
       array_attribute config_date="04/03/96"
       machine a_node
       machine_attribute aka="congo"
       hostname congo.engr.hitech.com
```

Tip: You can write code that fetches any arrayname, machine name, or attribute string from any node in the array.

Configuring Authentication Codes

In Array Services 3.5 only one type of authentication is provided: a simple numeric key that can be required with any Array Services command. You can specify a single authentication code number for each node. The user must specify the code with any command entered at that node, or addressed to that node using the `-s` option (see "Summary of Common Command Options" on page 14).

The `arshell` command is like `rsh` in that it runs a command on another machine under the userid of the invoking user. Use of authentication codes makes Array Services somewhat more secure than `rsh`.

Configuring Array Commands

The user can invoke arbitrary system commands on single nodes using the `arshell` command (see "Using arshell" on page 21). The user can also launch MPI and PVM programs that automatically distribute over multiple nodes. However, the only way to launch coordinated system programs on all nodes at once is to use the `array` command. This command does not accept any system command; it only permits execution of commands that the administrator has configured into the Array Services database.

You can define any set of commands that your users need. You have complete control over how any single Array node executes a command (the definition can be different in different nodes). A command can simply invoke a standard system command, or, since you can define a command as invoking a script, you can make a command arbitrarily complex.

Operation of Array Commands

When a user invokes the `array` command, the subcommand and its arguments are processed by the destination node specified by `-s`. Unless the `-l` option was given, that daemon also distributes the subcommand and its arguments to all other array nodes that it knows about (the destination node might be configured with only a subset of nodes). At each node, `arrayd` searches the configuration database for a COMMAND entry with the same name as the array subcommand.

In the following example, the subcommand `uptime` is processed by `arrayd` in node `tokyo`:

```
array -s tokyo uptime
```

When `arrayd` finds the subcommand valid, it distributes it to every node that is configured in the default array at node `tokyo`.

The COMMAND entry for `uptime` is distributed in this form (you can read it in the file `/usr/lib/array/arrayd.conf`).

```
command uptime          # Display uptime/load of all nodes in array
        invoke /usr/lib/array/auptime %LOCAL
```

The INVOKE subentry tells `arrayd` how to execute this command. In this case, it executes a shell script `/usr/lib/array/auptime` , passing it one argument, the name of the local node. This command is executed at every node, with %LOCAL replaced by that node's name.

Summary of Command Definition Syntax

Look at the basic set of commands distributed with Array Services 3.5 (`/usr/lib/array/arrayd.conf`). Each COMMAND entry is defined using the subentries shown in Table 2-7. (These are described in great detail in the man page `arrayd.conf`(4).)

Table 2-7 Subentries of a COMMAND Definition

Keyword	Meaning of Following Values
COMMAND	The name of the command as the user gives it to `array`.
INVOKE	A system command to be executed on every node. The argument values can be literals, or arguments given by the user, or other substitution values.
MERGE	A system command to be executed only on the distributing node, to gather the streams of output from all nodes and combine them into a single stream.
USER	The user ID under which the INVOKE and MERGE commands run. Usually given as USER %USER, so as to run as the user who invoked `array`.
GROUP	The group name under which the INVOKE and MERGE commands run. Usually given as GROUP %GROUP, so as to run in the group of the user who invoked `array` (see the `groups`(1) man page).
PROJECT	The project under which the INVOKE and MERGE commands run. Usually given as PROJECT %PROJECT, so as to run in the project of the user who invoked `array` (see the `projects`(5) man page).
OPTIONS	A variety of options to modify this command; see Table 2-9.

The system commands called by INVOKE and MERGE must be specified as full pathnames, because `arrayd` has no defined execution path. As with a shell script, these system commands are often composed from a few literal values and many substitution strings. The substitutions that are supported (which are documented in detail in the `arrayd.conf`(4) man page) are summarized in Table 2-8.

Table 2-8 Substitutions Used in a COMMAND Definition

Substitution	Replacement Value
%1..%9; %ARG(*n*); %ALLARGS; %OPTARG(*n*)	Argument tokens from the user's subcommand. %OPTARG does not produce an error message if the specified argument is omitted.
%USER, %GROUP, %PROJECT	The effective user ID, effective group ID, and project of the user who invoked `array`.
%REALUSER, %REALGROUP	The real user ID and real group ID of the user who invoked `array`.
%ASH	The ASH under which the INVOKE or MERGE command is to run.
%PID(*ash*)	List of PID values for a specified ASH. %PID(%ASH) is a common use.
%ARRAY	The array name, either default or as given in the `-a` option.
%LOCAL	The hostname of the executing node.
%ORIGIN	The full domain name of the node where the `array` command ran and the output is to be viewed.
%OUTFILE	List of names of temporary files, each containing the output from one node's INVOKE command (valid only in the MERGE subentry).

The OPTIONS subentry permits a number of important modifications of the command execution; these are summarized in Table 2-9.

Table 2-9 Options of the COMMAND Definition

Keyword	Effect on Command
LOCAL	Do not distribute to other nodes (effectively forces the `-l` option).
NEWSESSION	Execute the INVOKE command under a newly created ASH. %ASH in the INVOKE line is the new ASH. The MERGE command runs under the original ASH, and %ASH substitutes as the old ASH in that line.
SETRUID	Set both the real and effective user ID from the USER subentry (normally USER only sets the effective UID).
SETRGID	Set both the real and effective group ID from the GROUP subentry (normally GROUP sets only the effective GID).
QUIET	Discard the output of INVOKE, unless a MERGE subentry is given. If a MERGE subentry is given, pass INVOKE output to MERGE as usual and discard the MERGE output.
NOWAIT	Discard the output and return as soon as the processes are invoked; do not wait for completion (a MERGE subentry is ineffective).

Configuring Local Options

The LOCAL entry specifies options to `arrayd` itself. The most important options are summarized in Table 2-10.

Table 2-10 Subentries of the LOCAL Entry

Subentry	Purpose
DIR	Pathname for the `arrayd` working directory, which is the initial, current working directory of INVOKE and MERGE commands. The default is `/usr/lib/array`.
DESTINATION ARRAY	Name of the default array, used when the user omits the `-a` option. When only one ARRAY entry is given, it is the default destination.

Subentry	Purpose
USER, GROUP, PROJECT	Default values for COMMAND execution when USER, GROUP, or PROJECT are omitted from the COMMAND definition.
HOSTNAME	Value returned in this node by %LOCAL. Default is the hostname.
PORT	Socket to be used by `arrayd`.

If you do not supply LOCAL USER, GROUP, and PROJECT values, the default values for USER and GROUP are “guest.”

The HOSTNAME entry is needed whenever the `hostname` command does not return a node name as specified in the ARRAY MACHINE entry. In order to supply a LOCAL HOSTNAME entry unique to each node, each node needs an individualized copy of at least one configuration file.

Designing New Array Commands

A basic set of commands is distributed in the file `/usr/lib/array/arrayd.conf.template` . You should examine this file carefully before defining commands of your own. You can define new commands which then become available to the users of the Array system.

Typically, a new command will be defined with an INVOKE subentry that names a script written in sh, csh, or Perl syntax. You use the substitution values to set up arguments to the script. You use the USER, GROUP, PROJECT, and OPTIONS subentries to establish the execution conditions of the script. For one example of a command definition using a simple script, see "About the Distributed Example" on page 22.

Within the invoked script, you can write any amount of logic to verify and validate the arguments and to execute any sequence of commands. For an example of a script in Perl, see `/usr/lib/array/aps`, which is invoked by the `array ps` command.

Note: Perl is a particularly interesting choice for `array` commands, since Perl has native support for socket I/O. In principle at least, you could build a distributed application in Perl in which multiple instances are launched by `array` and coordinate and exchange data using sockets. Performance would not rival the highly tuned MPI and PVM libraries, but development would be simpler.

The administrator has need for distributed applications as well, since the configuration files are distributed over the Array. Here is an example of a distributed command to reinitialize the Array Services database on all nodes at once. The script to be executed at each node, called `/usr/lib/array/arrayd-reinit` would read as follows:

```
#!/bin/sh
# Script to reinitialize arrayd with a new configuration file
# Usage:  arrayd-reinit <hostname:new-config-file>
sleep 10      # Let old arrayd finish distributing
rcp $1 /usr/lib/array/
/etc/rc.d/init.d/array restart
exit 0
```

The script uses `rcp` to copy a specified file (presumably a configuration file such as `arrayd.conf`) into `/usr/lib/array` (this will fail if %USER is not privileged). Then the script restarts `arrayd` (see `/etc/rc.d/init.d/array`) to reread configuration files.

The command definition would be as follows:

```
command reinit
   invoke /usr/lib/array/arrayd-reinit %ORIGIN:%1
   user   %USER
   group  %GROUP
   options nowait   # Exit before restart occurs!
```

The INVOKE subentry calls the restart script shown above. The NOWAIT option prevents the daemon's waiting for the script to finish, since the script will kill the daemon.

CPU Memory Sets and Scheduling

This chapter describes the CPU memory sets and scheduling (CpuMemSet) application interface for managing system scheduling and memory allocation across the various CPUs and memory blocks in a system.

CpuMemSets provides a Linux kernel facility that enables system services and applications to specify on which CPUs they may be scheduled and from which nodes they may allocate memory. On an SGI Altix 3000 system, each C-brick contains two nodes. The default configuration makes all CPUs and all system memory available to all applications. The CpuMemSet facility can be used to restrict any process, process family, or process virtual memory region to a specified subset of the system CPUs and memory.

Any service or application with sufficient privilege may alter its cpumemset (either the set or map). The basic CpuMemSet facility requires root privilege to acquire more resources, but allows any process to remove (cease using) a CPU or memory node.

The CpuMemSet interface adds two layers called cpumemmap and cpumemset to the existing Linux scheduling and resource allocation code.

The lower cpumemmap layer provides a simple pair of maps that:

- Map system CPU numbers to application CPU numbers
- Map system memory block numbers to application block numbers

The upper cpumemset layer:

- Specifies on which application CPUs a process can schedule a task
- Specifies which application memory blocks the kernel or a virtual memory area can allocate

The CpuMemSet interface allows system administrators to control the allocation of a system CPU and of memory block resources to tasks and virtual memory areas. It allows an application to control the use of the CPUs on which its tasks execute and to obtain the optimal memory blocks from which its tasks's virtual memory areas obtain system memory.

The CpuMemSet interface provides support for such facilities as `dplace`(1), `runon`(1), cpusets, and nodesets.

The runon(1) command relies on CpuMemSets to enable you to run a specified command on a specified list of CPUs. Both a C shared library and Python language module are provided to access the CpuMemSets system interface. For more information on the runon command, see "Using the runon(1) Command" on page 44. For more information on the Python interface, see "Managing CpuMemSets" on page 45.

This chapter describes the following topics:

Memory Management Terminology

The primitive concepts that are discussed in this chapter are hardware processors (CPUs) and system memory and their corresponding software constructs of tasks and virtual memory areas.

System Memory Blocks

On a nonuniform memory access (NUMA) system, blocks are the equivalence classes of main memory locations defined by the relation of distance from CPUs. On a typical symmetric multiprocessing (SMP) or uniprocessing (UP) system, all memory is the same distance from any CPU (same speed), and equivalent for the purposes of this discussion. System memory blocks do not include special purpose memory, such as I/O and video frame buffers, caches, peripheral registers, and I/O ports.

Tasks

Tasks are execution threads that are part of a process. They are scheduled on hardware processors called CPUs.

The Linux kernel schedules threads of execution it calls *tasks*. A task executes on a single processor (CPU) at a time. At any point in time, a task may be:

- Waiting for some event or resource or interrupt completion
- Executing on a CPU. Tasks may be restricted from executing on certain CPUs.

Linux kernel tasks execute on CPU hardware processors. This does not include special purpose processors, such as direct memory access (DMA) engines, vector processors, graphics pipelines, routers, or switches.

Virtual Memory Areas

For each task, the Linux kernel keeps track of multiple virtual address regions called virtual memory areas. Some virtual memory areas may be shared between multiple tasks. The kernel memory management software manages virtual memory areas in units of pages. Each given page in the address space of a virtual memory area may be as follows:

- Not yet allocated
- Allocated but swapped out to disk
- Currently residing in allocated system memory

Virtual memory areas may be restricted from allocating memory blocks from certain system memory blocks.

Nodes

Typically, NUMA systems consists of nodes. Each node contains a number of CPUs and system memory. On an SGI Altix 3000 system, for example, each C-brick contains two nodes. The CpuMemSet system focuses on CPUs and memory blocks, not on nodes. For currently available SGI systems, the CPUs and all memory within a node are equivalent.

CpuMemSet System Implementation

The CpuMemSet system is implemented by two separate layers as follows:

- "Cpumemmap" on page 40

- "cpumemset" on page 40

Cpumemmap

The lower layer —cpumemmap (`cmm`)— provides a simple pair of maps that map system CPU and memory block numbers to application CPU and memory block numbers. *System numbers* are used by the kernel task scheduling and memory allocation code, and typically are assigned to all CPUs and memory blocks in the system. *Application numbers* are assigned to the CPUs and memory blocks in an application's cpumemset and are used by the application to specify its CPU and memory affinity for the CPUs and memory blocks it has available in its cpumemmap. Each process, each virtual memory area, and the kernel has such a cpumemmap. These maps are inherited across `fork` calls, `exec` calls, and the various ways to create virtual memory areas. Only a process with root privileges can extend a cpumemmap to include additional system CPUs or memory blocks. Changing a map causes kernel scheduling code to immediately start using the new system CPUs and causes kernel allocation code to allocate additional memory pages using the new system memory blocks. Memory already allocated on old blocks is not migrated, unless some non-CpuMemSet mechanism is used.

The cpumemmaps do not have holes. A given cpumemmap of size *n*, maps all application numbers between 0 and *n*–1, inclusively, to valid system numbers. An application can rely on any CPU or memory block numbers known to it to remain valid. However, cpumemmaps are not necessarily one-to-one (injective). Multiple application numbers can map to the same system number.

When a `cmsSetCMM`() routine is called, changes to cpumemmaps are applied to system masks, such as `cpus_allowed`, and lists, such as zone lists, used by existing Linux scheduling and allocation software.

cpumemset

The upper cpumemset (`cms`) layer specifies the application CPUs on which a process can schedule a task to execute. It also specifies application memory blocks, known to the kernel or a virtual memory area, from which it can allocate memory blocks. A different list is specified for each CPU that may execute the request. An application may change the cpumemset of its tasks and virtual memory areas. A root process can change the cpumemset used for kernel memory allocation. A root process can change the cpumemsets of any process. Any process may change the cpumemsets of other processes with the same user ID (UID)(`kill`(2) permissions), except that the current

implementation does not support changing the cpumemsets attached to the virtual memory areas of another process.

Each task has two cpumemsets. One cpumemset defines the task's current CPU allocation and created virtual memory areas. The other cpumemset is inherited by any child process the task forks. Both the current and child cpumemsets of a newly forked process are set to copies of the child cpumemset of the parent process. Allocations of memory to existing virtual memory areas visible to a process depend on the cpumemset of that virtual memory area (as acquired from its creating process at creation, and possibly modified since), not on the cpumemset of the currently accessing task.

During system boot, the kernel creates and attaches a default cpumemmap and cpumemset that are used everywhere on the system. By default, this initial map and cpumemset contain all CPUs and all memory blocks.

An optional kernel-boot command line parameter causes this initial cpumemmap and cpumemset to contain only the first CPU and one memory block, rather than all of them, as follows:

```
cpumemset_minimal=1
```

This is for the convenience of system management services that are designed to take greater control of the system.

The kernel schedules a task only on the CPUs in the task's cpumemset, and allocates memory only to a user virtual memory area, chosen from the list of memories in the memory list of that area. The kernel allocates kernel memory only from the list of memories in the cpumemset attached to the CPU that is executing the allocation request, except for specific calls within the kernel that specify some other CPU or memory block.

Both the current and child cpumemmaps and cpumemsets of a newly forked process are taken from the child settings of its parent process. Memory allocated during the creation of the new process is allocated according to the child cpumemset of the parent process and associated cpumemmap because that cpumemset is acquired by the new process and then by any virtual memory area created by that process.

The cpumemset (and associated cpumemmap) of a newly created virtual memory area is taken from the current cpumemset of the task creating it. In the case of attaching to an existing virtual memory area, the scenario is more complicated. Both memory mapped memory objects and UNIX System V shared memory regions can be attached to by multiple processes, or even attached to multiple times by the same process at different addresses. If such an existing memory region is attached to, then

by default the new virtual memory area describing that attachment inherits the current cpumemset of the attaching process. If, however, the policy flag `CMS_SHARE` is set in the cpumemset currently linked to from each virtual memory area for that region, then the new virtual memory area is also linked to this same cpumemset.

When allocating another page to an area, the kernel chooses the memory list for the CPU on which the current task is being executed, if that CPU is in the cpumemset of that memory area, otherwise it chooses the memory list for the default CPU (see `CMS_DEFAULT_CPU`) in that memory area's cpumemset. The kernel then searches the chosen memory list, looking for available memory. Typical kernel allocation software searches the same list multiple times, with increasingly aggressive search criteria and memory freeing actions.

The cpumemmap and cpumemset calls with the `CMS_VMAREA` flag apply to all future allocation of memory by any existing virtual memory area, for any pages overlapping any addresses in the range [start, start+len). This is similar to the behavior of the `madvise`, `mincore`, and `msync` functions.

Installing, Configuring, and Tuning CpuMemSets

This section describes how to install, configure, and tune CpuMemSets on your system and contains the following topics:

Installing CpuMemSets

The CpuMemSets facility is automatically included in SGI ccNUMA Linux systems, including the kernel support; the user level library (`libcpumemsets.so`) used to access this facility from C language programs; a Python module (`cpumemsets`) for access from a scripting environment; and a `runon`(1) command for controlling which CPUs and memory nodes an application may be allowed to use.

To use the Python interface, from a script perform the following:

```
import cpumemsets
print cpumemsets.__doc__
```

Configuring CpuMemSets

No configuration is required. All processes, all memory regions, and the kernel are automatically provided with a default CpuMemSet, which includes all CPUs and memory nodes in the system.

Tuning CpuMemSets

You can change the default CpuMemSet to include only the first CPU and first memory node by providing this additional option on the kernel boot command line (accessible via `elilo`) as follows:

```
cpumemset_minimal=1
```

This is useful if you want to dedicate portions of your system CPUs or memory to particular tasks.

Using CpuMemSets

This section describes how CpuMemSets are used on your system and contains the following topics:

Using the `runon`(1) Command

The `runon`(1) command allows you to run a command on a specified list of CPUs. The syntax of the command is as follows:

```
runon cpu ... command [args ...]
```

The `runon` command, shown in Example 3-1, executes a command, assigning the command to run only on the listed CPUs. The list of CPUs may include individual CPUs or an inclusive range of CPUs separated by a hyphen. The specified CPU affinity is inherited across `fork`(2) and `exec`(2) system calls. All options are passed in the `argv` list to the executable being run.

Example 3-1 Using the `runon`(1) Command

To execute the `echo`(1) command on CPUs 1, 3, 4, 5, or 9, perform the following:

```
runon 1 3-5 9 echo Hello World
```

For more information, see the `runon`(1) man page.

Initializing CpuMemSets

Early in the boot sequence, before the normal kernel memory allocation routines are usable, the kernel sets up a single default cpumemmap and cpumemset. If no action is ever taken by user level code to change them, this one map and one set applies to the kernel and all processes and virtual memory areas for the life of that system boot.

By default, this map includes all CPUs and memory blocks, and this set allows scheduling on all CPUs and allocation on all blocks.

An optional kernel boot parameter causes this initial map and set to include only one CPU and one memory block, in case the administrator or some system service will be managing the remaining CPUs and blocks in some specific way.

As soon as the system has booted far enough to run the first user process, `init`(1M), an early `init` script may be invoked that examines the topology and metrics of the system, and establishes optimized cpumemmap and cpumemset settings for the kernel and for the `init` process. Prior to that, various kernel daemons are started and kernel data structures are allocated, which may allocate memory without the benefit of these optimized settings. This reduces the amount of information that the kernel needs about special topology and distance attributes of a system in that the kernel needs only enough information to get early allocations placed correctly. More detailed topology information can be kept in the user application space.

Operating on CpuMemSets

On a system supporting CpuMemSets, all processes have their scheduling constrained by their cpumemmap and cpumemset. The kernel will not schedule a process on a CPU that is not allowed by its cpumemmap and cpumemset. The Linux task scheduler must support a mechanism, such as the `cpus_allowed` bit vector, to control on which CPUs a task may be scheduled.

Similarly, all memory allocation is constrained by the cpumemmap and cpumemset associated to the kernel or virtual memory area requesting the memory, except for specific requests within the kernel. The Linux page allocation code has been changed to search only in the memory blocks allowed by the virtual memory area requesting memory. If memory is not available in the specified memory blocks, the allocation fails or sleeps, awaiting memory. The search for memory does not consider other memory blocks in the system.

It is this "mandatory" nature of cpumemmaps and cpumemsets that allows CpuMemSets to provide many of the benefits of hard partitioning in a dynamic, single-system, image environment (see "Hard Partitioning versus CpuMemSets" on page 47).

Managing CpuMemSets

System administrators and services with root privileges manage the initial allocation of system CPUs and memory blocks to cpumemmaps, deciding which applications will be allowed the use of specified CPUs and memory blocks. They also manage the cpumemset for the kernel, which specifies what order to use to search for kernel memory, depending on which CPU is executing the request.

Almost all ordinary applications will be unaware of CpuMemSets, and will run in whatever CPUs and memory blocks their inherited cpumemmap and cpumemset dictate.

Large multiprocessor applications can take advantage of CpuMemSets by using existing legacy application programming interfaces (APIs) to control the placement of the various processes and memory regions that the application manages. Emulators for whatever API the application is using can convert these requests into cpumemset changes, which then provide the application with detailed control of the CPUs and memory blocks provided to the application by its cpumemmap.

To alter default cpumemsets or cpumemmaps, use one of the following:

- The C language interface provided by the library (`libcpumemsets`)

- The Python interface provided by the module (`cpumemsets`)
- The `runon`(1) command

Initializing System Service on CpuMemSets

The cpumemmaps do not have system-wide names; they cannot be created ahead of time when a system is initialized, and then attached to later by name. The cpumemmaps are like classic UNIX anonymous pipes or anonymous shared memory regions, which are identifiable within an individual process by file descriptor or virtual address, but not by a common namespace visible to all processes on the system.

When a boot script starts up a major service on some particular subset of the machine (its own cpumemmap), the script can set its child map to the cpumemmap desired for the major service it is spawning and then invoke `fork` and `exec` calls to execute the service. If the service has root privilege, it can extend its own cpumemmaps, as determined by the system administrator.

A higher level API can use CpuMemSets to define a virtual system that could include a certain number of CPUs and memory blocks and the means to manage these system resources.

A daemon with root privilege can run and be responsible for managing the virtual systems defined by the API; or perhaps some daemon without root privilege can run with access to all the CPUs and memory blocks that might be used for this service.

When some user process application is granted permission by the daemon to run on the named virtual systems, the daemon sets its child map to the cpumemmap describing the CPU and memory available to that virtual system and spawns the requested application on that map.

Resolving Pages for Memory Areas

The cpumemmap and cpumemset calls that specify a range of memory (`CMS_VMAREA`) apply to all pages in the specified range. The internal kernel data structures, tracking each virtual memory area in an address space, are automatically split if a cpumemmap or cpumemset is applied to only part of the range of pages in that virtual memory area. This splitting happens transparently to the application. Subsequent re-merging of two such neighboring virtual memory areas may occur if the two virtual memory areas no longer differ. This same behavior is seen in the system calls `madvise`(2), `msync`(2), and `mincore`(2).

Determining an Application's Current CPU

The `cmsGetCpu`() function returns the currently executing application CPU number as found in the cpumemmap of the current process. This information, along with the results of the `cmsQuery*()` calls, may be helpful for applications running on some architectures to determine the topology and current utilization of a system. If a process can be scheduled on two or more CPUs, the results of `cmsGetCpu`() may become invalid even before the query returns to the invoking user code.

Determining the Memory Layout of cpumemmaps and cpumemsets

The `cmsQuery*`() library calls construct cpumemmaps and cpumemsets by using `malloc`(3) to allocate each distinct structure and array element in the return value and linking them together. The `cmsFree*`() calls assume this layout, and call the `free`(3) routine on each element.

If you construct your own cpumemmap or cpumemset, using some other memory layout, do not pass that layout to the `cmsFree*`() call.

You may alter in place and replace malloc'd elements of a cpumemmap or cpumemset returned by a `cmsQuery*`() call, and pass the result back into a corresponding `cmsSet*`() or `cmsFree*`() call.

Hard Partitioning versus CpuMemSets

On a large NUMA system, you may want to control which subset of processors and memory is devoted to s specified major application. This can be done using "hard" partitions, where subsets of the system are booted using separate system images and the partitions act as a cluster of distinct computers rather than a single-system-image computer.

Partitioning a large NUMA system partially defeats the advantages of a large NUMA machine with a single system image. CpuMemSets enable you to carve out more flexible, possibly overlapping, partitions of the system's CPUs and memory. This allows all processes to see a single system image, without rebooting, but guarantees certain CPU and memory resources to selected applications at various times.

SGI software partitioning technology overcomes many of the disadvantages of hard partitioning. A single SGI ProPack for Linux server can be divided into multiple distinct systems, each with its own console, root filesystem, and IP network address. Each of these software-defined group of processors are distinct systems referred to as

a partition. Each partition can be rebooted, loaded with software, powered down, and upgraded independently. The partitions communicate with each other over an SGI NUMAlink connection. Collectively, all of these partitions compose a single, shared-memory cluster.

Direct memory access between partitions, sometimes referred to as global shared memory, is made available by the XPC and XPMEM kernel modules. This allows processes in one partition to access physical memory located on another partition. The benefits of global shared memory are currently available via SGI's Message Passing Toolkit (MPT) software.

CpuMemSets provide you with substantial control over system processor and memory resources without the attendant inflexibility of hard partitions.

It is relatively easy to configure a large SGI Altix system into partitions and reconfigure the machine for specific needs. No cable changes are needed to partition or repartition an SGI Altix machine. Partitioning is accomplished by commands sent to the system controller. For information on how to partition a system, see "System Partitioning" in the *Linux Configuration and Operations Guide*. For details on system controller commands, see SGI L1 and L2 Controller Software User's Guide.

Error Messages

This section describes typical error situations. Some of them are as follows:

- If a request is made to set a cpumemmap that has fewer CPUs or memory blocks listed than needed by any cpumemsets that will be using that cpumemmap after the change, the `cmsSetCMM()` call fails, with `errno` set to `ENOENT`. You cannot remove elements of a cpumemmap that are in use.
- If a request is made to set a cpumemset that references CPU or memory blocks not available in its current cpumemmap, the `cmsSetCMS()` call fails, with `errno` set to `ENOENT`. You cannot reference unmapped application CPUs or memory blocks in a cpumemset.
- If a request is made without root privileges to set a cpumemmap by a process , and that request attempts to add any system CPU or memory block number not currently in the map being changed, the request fails, with `errno` set to `EPERM`.
- If a `cmsSetCMS()` request is made on another process, the requesting process must either have root privileges, or the real or effective user ID of the sending process must equal the real or saved set-user-ID of the other process, or else the

request fails, with `errno` set to `EPERM`. These permissions are similar to those required by the `kill`(2) system call.

- Every cpumemset must specify a memory list for the `CMS_DEFAULT_CPU`, to ensure that regardless of which CPU a memory request is executed on, a memory list will be available to search for memory. Attempts to set a cpumemset without a memory list specified for the `CMS_DEFAULT_CPU` fail, with `errno` set to `EINVAL`.
- If a request is made to set a cpumemset that has the same CPU (application number) listed in more than one array `cpus` of CPUs sharing any `cms_memory_list_t` structures, then the request fails, with `errno` set to `EINVAL`. Otherwise, duplicate CPU or memory block numbers are harmless, except for minor inefficiencies.
- The operations to query and set cpumemmaps and cpumemsets can be applied to any process ID (PID). If the PID is zero, then the operation is applied to the current process. If the specified PID does not exist, then the operation fails, with `errno` set to `ESRCH`.

Cpuset System

The Cpuset System is primarily a workload manager tool permitting a system administrator to restrict the number of processors that a process or set of processes may use.

In Linux, when a process running on a cpuset runs out of available memory on the requested nodes, memory on other nodes can be used. The `MEMORY_LOCAL` policy is the policy that supports using memory on other nodes if no memory is freely available on the requested nodes and currently is the only policy supported.

A system administrator can use cpusets to create a division of CPUs within a larger system. Such a divided system allows a set of processes to be contained to specific CPUs, reducing the amount of interaction and contention those processes have with other work on the system. In the case of a restricted cpuset, the processes that are attached to that cpuset will not be affected by other work on the system; only those processes attached to the cpuset can be scheduled to run on the CPUs assigned to the cpuset. An open cpuset can be used to restrict processes to a set of CPUs so that the effect these processes have on the rest of the system is minimized. In Linux the concept of restricted is essentially cooperative, and can be overriden by processes with root privilege.

The state files for a cpuset reside in the `/var/cpuset` directory.

When you boot your system, an `init` script called `cpunodemap` creates a boot cpuset that by default contains all the CPUs in the system; enabling any process to run on any CPU and use any system memory. Processes on a Linux system run on the entire system unless they are placed on a specific cpuset or are constrained by some other tool.

A system administrator might choose to use cpusets to divide a system into two halves, with one half supporting normal system usage and the other half dedicated to a particular application. You can make the changes you want to your cpusets and all new processes attached to those cpusets will adhere to the new settings. The advantage this mechanism has over physical reconfiguration is that the configuration may be changed using the cpuset system and does not need to be aligned on a hardware module boundary.

Static cpusets are defined by an administrator after a system had been started. Users can attach processes to these existing cpusets. The cpusets continue to exist after jobs are finished executing.

Dynamic cpusets are created by a workload manager when required by a job. The workload manager attaches a job to a newly created cpuset and destroys the cpuset when the job has finished executing.

The `runon`(1) command allows you to run a command on a specified list of CPUs. If you use the `runon` command to restrict a process to a subset of CPUs that it is already executing on, `runon` will restrict the process without root permission or the use of cpusets. If the you use the `runon` command to run a command on different or additional CPUs, `runon` invokes the `cpuset` command to handle the request. If all of the specified CPUs are within the same cpuset and you have the appropriate permissions, the `cpuset` command will execute the request.

The cpuset library provides interfaces that allow a programmer to create and destroy cpusets, retrieve information about existing cpusets, obtain the properties associated with a cpuset, and to attach a process and all of its children to a cpuset.

The bootcpuset facility provides a method to restrict all normal start-up processes (including `init` and its descendents) to some portion of the machine and allow specific users to use the other portion of the machine for their special purpose applications. For more information on the bootcpuset facility, see "Bootcpuset" on page 54.

This chapter contains the following sections:

Cpusets on Linux versus IRIX

This sections describes the major differences between how the Cpuset System is implemented on the SGI ProPack for Linux releases versus the current IRIX operating system. These differences are likely to change for future releases of SGI ProPack for Linux.

Major differences include the following:

- In IRIX, the cpuset command maintains the `/etc/cpusettab` file that defines the currently established cpusets, including the boot cpuset. In Linux, state files for cpusets are maintained in a directory called `/var/cpuset`.
- Permission checking against the cpuset configuration file permissions is not implemented for this Linux release. For more information, see "Cpuset Configuration File" on page 60.
- In Linux, you can use `cpumemset_minimal` boot parameter to keep the `init` process (and the shell and shared libraries that early boot `init` scripts load) constrained to the first node as a means to control usage of the system. For more information, see "Bootcpuset" on page 54.
- Linux currently supports only the `MEMORY_LOCAL` policy that allows a process to obtain memory on other nodes if memory is not freely available on the requested nodes. For more information on Cpuset policies, see "Cpuset Configuration File" on page 60.
- Linux does not support the `MEMORY_EXCLUSIVE` policy.

  The `MEMORY_EXCLUSIVE` policy and the related notion of a "restricted" cpuset are essentially only cooperative in Linux, rather than mandatory. On Linux, a process with root privilege may use CpuMemSet calls directly to run tasks on any CPU and use any memory, potentially violating cpuset boundaries and exclusiveness. For more information on CpuMemSets, see Chapter 3, "CPU Memory Sets and Scheduling" on page 37.
- In IRIX, a cpuset can only be destroyed using the `cpusetDestroy` function if there are no processes currently attached to the cpuset. In Linux, when a cpuset is destroyed using the `cpusetDestroy` function, processes currently running on the cpuset continue to run and can spawn a new process that will continue to run on the cpuset. Otherwise, new processes are not allowed to run on the cpuset.

- The current Linux release does not support the cpuset library routines, `cpusetMove`(3x) and `cpusetMoveMigrate`(3x), that can be used to move processes between cpusets and optionally migrate their memory.
- In IRIX, the `runon`(1) command cannot run a command on a CPU that is part of a cpuset unless the user has write or group write permission to access the configuration file of the cpuset. On Linux, this restriction is not implemented for this release.

Bootcpuset

A bootcpuset consists of a number of nodes, specified by a system administrator, on which user-level processes and memory are constrained. User-level processes will not run on the remaining nodes in the system, unless placed there by commands or system calls, such as, `runon`(1), `dplace`(1), `cpuset`(1), or `cpumemsets`. User process scheduling is tightly constrained to the CPUs on the bootcpuset nodes. Memory allocation for user space is preferentially allocated from the bootcpuset nodes but not tightly constrained in the current implementation. If the nodes in the bootcpuset are short of free memory, the requests for memory may be met by taking memory from other nodes.

The `bootcpuset.conf`(5) file specifies the number of nodes to be included in the bootcpuset. The `bootcpuset.rc`(8) `init` script uses the `bootcpuset`(8) command to constrain the `init` process and its descendents to the CPUs and memory on these nodes. For more information, see the `bootcpuset`(8), `bootcpuset.rc`(8), and (5) man pages.

To configure the bootcpuset to be created automatically during system startup, use the `chkconfig`(8) command as follows:

```
chkconfig --add bootcpuset
```

To stop the bootcpuset from being created automatically during system startup, use the `chkconfig`(8) command as follows:

```
chkconfig --del bootcpuset
```

For more information on the `chkconfig` command, see the `chkconfig`(8) man page.

If you plan to use the bootcpuset facility, SGI advises that you also boot your system with the kernel boot parameter `cpumemset_minimal=1` (accessible via `elilo`), to keep the `init` process (and the shell and shared libraries that early boot `init` scripts

load) constrained to the first node, prior to the point that the `bootcpuset.rc init` script executes.

For more information on kernel boot command line options, see "cpumemset" on page 40 and "Tuning CpuMemSets" on page 43.

Using Cpusets

This section describes the basic steps for using cpusets and the `cpuset`(1) command. For a detailed example, see "Cpuset System Examples" on page 57.

To install the Cpuset System software, see "Installing the Cpuset System" on page 63.

To use cpusets, perform the following steps:

1. Create a cpuset configuration file and give it a name. For the format of this file, see "Cpuset Configuration File" on page 60. For restrictions that apply to CPUs belonging to cpusets, see "Restrictions on CPUs within Cpusets" on page 57.
2. Create the cpuset with the configuration file specified by the `-f` parameter and the name specified by the `-q` parameter.

The `cpuset`(1) command is used to create and destroy cpusets, to retrieve information about existing cpusets, and to attach a process and all of its children to a cpuset. The syntax of the `cpuset` command is as follows:

```
cpuset [-q cpuset_name[,cpuset_name_dest][setName -1][-A command]
[-c -f filename] [-d] [-l] [-m] [-Q] [-C] [-h]
```

The `cpuset` command accepts the following options:

`-q` *cpuset_name* [`-A` *command*]

Runs the specified command on the cpuset identified by the `-q` parameter. If the user does not have access permissions or the cpuset does not exist, an error is returned.

Note: File permission checking against the configuratuion file permissions is not implemented for this release of SGI Linux.

`-q` *cpuset_name* [`-c -f` *filename*]	Creates a cpuset with the configuration file specified by the `-f` parameter and the name specified by the `-q` parameter. The operation fails if the cpuset name already exists, a CPU specified in the cpuset configuration file is already a member of a cpuset, or the user does not have the requisite permissions. **Note:** File permission checking against the configuratuion file permissions is not implemented for this release of SGI Linux.
`-q` *cpuset_name* `-d`	Destroys the specified cpuset. Any processes currently attached to it continue running where they are, but no further commands to list (`-Q`) or attach (`-A`) to that cpuset will succeed.
`-q` *cpuset_name* `-Q`	Prints a list of the CPUs that belong to the cpuset.
`-q` *set_Name* `-l`	Lists all processes in a cpuset.
`-C`	Prints the name of the cpuset to which the process is currently attached.
`-Q`	Lists the names of all the cpusets currently defined.
`-h`	Print the command's usage message.

3. Execute the `cpuset` command to run a command on the cpuset you created as follows:

   `cpuset -q` *cpuset_name* `-A` *command*

For more information on using cpusets, see the `cpuset`(1) man page, "Restrictions on CPUs within Cpusets" on page 57, and "Cpuset System Examples" on page 57.

Restrictions on CPUs within Cpusets

The following restrictions apply to CPUs belonging to cpusets:

- A CPU should belong to only one cpuset.
- Only the superuser can create or destroy cpusets.
- The `runon`(1) command cannot run a command on a CPU that is part of a cpuset unless the user has write or group write permission to access the configuration file of the cpuset. (This restriction is not implemented for this release).

The Linux kernel does not enforce cpuset restriction directly. Rather restriction is established by booting the kernel with the optional boot command line parameter `cpumemset_minimal` that establishes the `CpuMemSets` initial kernel CpuMemSet to only include the first CPU and memory node. The rest of the systems CPUs and memory then remain unused until attached to using cpuset or some other facility with root privilege. The cpuset command and library support ensure restriction among clients of cpusets, but not from other processes.

For a description of `cpuset` command arguments and additional information, see the `cpuset`(1), `cpuset`(4), and `cpuset`(5) man pages.

Cpuset System Examples

This section provides some examples of using cpusets. This following specification creates a cpuset containing 8 CPUs and a cpuset containg 4 CPUs and will restrict those CPUs to running threads that have been explicitly assigned to the cpuset. Jobs running on the cpuset will use memory from nodes containing the CPUs in the cpuset. Jobs running on other cpusets or on the global cpuset will not use memory from these nodes.

Example 4-1 Creating Cpusets and Assigning Applications

Perform the following steps to create two cpusets on your system called `cpuset_art` and `cpuset_numberic`.

1. Create a dedicated cpuset called `cpuset_art` and assign a specific application, in this case, `gimp`, a GNU Image Manipulation Program, to run on it. Perform the following steps to accomplish this:

   a. Create a cpuset configuration file called `cpuset_1` with the following contents:

   ```
   # the cpuset configuration file called cpuset_1 that shows
   # a cpuset dedicated to a specific application
   MEMORY_LOCAL

   CPU 4-7
   CPU 8
   CPU 9
   CPU 10
   CPU 11
   ```

   Note: You can designate more than one CPU or a range of CPUs on a single line in the cpuset configuration file. In this example, you can designate CPUs 4 through 7 on a single line as follows: `CPU 4-7`. For more information on the cpuset configuration file, see "Cpuset Configuration File" on page 60.

   For an explanation of the `MEMORY_LOCAL` flag, see "Cpuset Configuration File" on page 60.

   b. Use the `chmod`(1) command to set the file permissions on the `cpuset_1` configuration file so that only members of group `artists` can execute the application `gimp` on the `cpuset_art` cpuset.

   c. Use the `cpuset`(1) command to create the `cpuset_art` cpuset with the configuration file `cpuset_1` specified by the `-c` and `-f` parameters and the name `cpuset_art` specified by the `-q` parameter.

   ```
   cpuset -q cpuset_art -c -f cpuset_1
   ```

   d. Execute the `cpuset` command as follows to run `gimp` on a dedicated cpuset:

   ```
   cpuset -q cpuset_art -A gimp
   ```

   The `gimp` job threads will run only on CPUs in this cpuset. `gimp` jobs will use memory from system nodes containing the CPUs in the cpuset. Jobs running on other cpusets will not use memory from these nodes. You could

use the `cpuset` command to run additional applications on the same cpuset using the syntax shown in this example.

2. Create a second cpuset file called `cpuset_number` and specify an application that will run only on this cpuset. Perform the following steps to accomplish this:

   a. Create a cpuset configuration file called `cpuset_2` with the following contents:

   ```
   # the cpuset configuration file called cpuset_2 that shows
   # a cpuset dedicated to a specific application
   EXCLUSIVE
   MEMORY_LOCAL

   CPU 12
   CPU 13
   CPU 14
   CPU 15
   ```

   For an explanation of the `EXCLUSIVE` flag, see "Cpuset Configuration File" on page 60.

   b. Use the `chmod`(1) command to set the file permissions on the `cpuset_2` configuration file so that only members of group `accountants` can execute the application `gnumeric` on the `cpuset_number` cpuset.

   c. Use the `cpuset`(1) command to create the `cpuset_number` cpuset with the configuration file `cpuset_2` specified by the `-c` and `-f` parameters and the name specified by the `-q` parameter.

   ```
   cpuset -q cpuset_number -c  -f cpuset_2
   ```

   d. Execute the `cpuset`(1) command as follows to run `gnumeric` on CPUs in the `cpuset_number` cpuset.

   ```
   cpuset -q cpuset_number -A gnumeric
   ```

   The `gnumeric` job threads will run only on this cpuset. `gnumeric` jobs will use memory from system nodes containing the CPUs in the cpuset. Jobs running on other cpusets will not use memory from these nodes.

You can create a bootcpuset and assign all system daemons and user logins to run on a single CPU leaving the rest of the system CPUs to be assigned to job specific cpusets. You can use the bootcpuset facility to create a bootcpuset using the `chkconfig --add bootcpuset` command. For more information, see "Bootcpuset" on page 54.

Cpuset Configuration File

This section describes the `cpuset`(1) command and the cpuset configuration file.

A cpuset is defined by a cpuset configuration file and a name. See the `cpuset`(4) man page for a definition of the file format. The cpuset configuration file is used to list the CPUs that are members of the cpuset. It also contains any additional arguments required to define the cpuset. A cpuset name is between 3 and 8 characters long; names of 2 or fewer characters are reserved. You can designate one or more CPUs or a range of CPUs as part of a cpuset on a single line in the cpuset configuration file. CPUs in a cpuset do **not** have to be specified in a particular order. Each cpuset on your system must have a separate cpuset configuration file.

Note: In a CXFS cluster environment, the cpuset configuration file should reside on the root file system. If the cpuset configuration file resides on a file system other than the root file system and you attempt to unmount the file system, the vnode for the cpuset remains active and the `unmount` command fails. For more information, see the `mount`(1M) man page.

The file permissions of the configuration file define access to the cpuset. When permissions need to be checked, the current permissions of the file are used. It is therefore possible to change access to a particular cpuset without having to tear it down and recreate it, simply by changing the access permission. Read access allows a user to retrieve information about a cpuset, while execute permission allows a user to attach a process to the cpuset.

Note: Permission checking against the cpuset configuration file permissions is not implemented for this release of SGI Linux.

By convention, CPU numbering on SGI systems ranges between zero and the number of processors on the system minus one.

The following is a sample configuration file that describes an exclusive cpuset containing three CPUs:

```
# cpuset configuration file
EXCLUSIVE
MEMORY_LOCAL
MEMORY_EXCLUSIVE

CPU 1
```

```
CPU 5
CPU 10
```

This specification will create a cpuset containing three CPUs. When the EXCLUSIVE flag is set, it restricts those CPUs to running threads that have been explicitly assigned to the cpuset. When the MEMORY_LOCAL flag is set, the jobs running on the cpuset will use memory from the nodes containing the CPUs in the cpuset. When the MEMORY_EXCLUSIVE flag is set, jobs running on other cpusets or on the global cpuset will normally not use memory from these nodes.

Note: For this Linux release, MEMORY_EXCLUSIVE, MEMORY_KERNEL_AVOID, MEMORY_MANDATORY, POLICY_PAGE, and POLICY_KILL are policies are not supported.

The following is a sample configuration file that describes an exclusive cpuset containing seven CPUs:

```
# cpuset configuration file
EXCLUSIVE
MEMORY_LOCAL
MEMORY_EXCLUSIVE

CPU 16
CPU 17-19, 21
CPU 27
CPU 25
```

Commands are newline terminated; characters following the comment delimiter, #, are ignored; case matters; and tokens are separated by whitespace, which is ignored.

The valid tokens are as follows:

Valid tokens	Description
EXCLUSIVE	Defines the CPUs in the cpuset to be restricted. It can occur anywhere in the file. Anything else on the line is ignored.
MEMORY_LOCAL	Threads assigned to the cpuset will attempt to assign memory only from nodes within the cpuset. Assignment of memory from outside the cpuset will occur only if no free memory is available from within

	the cpuset. No restrictions are made on memory assignment to threads running outside the cpuset.
`MEMORY_EXCLUSIVE`	Threads not assigned to the cpuset will not use memory from within the cpuset unless no memory outside the cpuset is available. When a cpuset is created and memory is occupied by threads that are already running on the cpuset nodes, no attempt is made to explicitly move this memory. If page migration is enabled, the pages will be migrated when the system detects the most references to the pages that are nonlocal.
`MEMORY_KERNEL_AVOID`	The kernel will attempt to avoid allocating memory from nodes contained in this cpuset. If kernel memory requests cannot be satisfied from outside this cpuset, this option will be ignored and allocations will occur from within the cpuset.
`MEMORY_MANDATORY`	The kernel will attempt to avoid allocating memory from nodes contained in this cpuset. If kernel memory requests cannot be satisfied from outside this cpuset, this option will be ignored and allocations will occur from within the cpuset.
`POLICY_PAGE`	Requires `MEMORY_MANDATORY`. This is the default policy if no policy is specified. This policy will cause the kernel to page user pages to the swap file to free physical memory on the nodes contained in this cpuset. If swap space is exhausted, the process will be killed.
`POLICY_KILL`	Requires `MEMORY_MANDATORY`. The kernel will attempt to free as much space as possible from kernel heaps, but will not page user pages to the swap file. If all physical memory on the nodes contained in this cpuset are exhausted, the process will be killed.

`CPU`	Specifies that a CPU will be part of the cpuset. The user can mix a single cpu line with a cpu list line. For example:
	
	`CPU 2`
	`CPU 3-4,5,7,9-12`

Installing the Cpuset System

The following steps are required to enable cpusets:

1. Configure the cpusets on across system reboots by using the `chkconfig`(8) utility as follows:

   ```
   chkconfig --add cpuset
   ```

2. To turn on cpusets, perform the following:

   ```
   /etc/rc.d/init.d/cpuset start
   ```

   This step will be done automatically for subsequent system reboots when the Cpuset System is configured on via the `chkconfig`(8) utility.

The following steps are required to disable cpusets:

1. To turn off cpusets, perform the following:

   ```
   /etc/rc.d/init.d/cpuset stop
   ```

2. To stop cpusets from initiating after a system reboot, use the `chkconfig`(8) command:

   ```
   chkconfig --del cpuset
   ```

Using the Cpuset Library

The cpuset library provides interfaces that allow a programmer to create and destroy cpusets, retrieve information about existing cpusets, obtain the properties associated with an existing cpuset, and to attach a process and all of its children to a cpuset. For more information on the Cpuset Library, see the `cpuset`(5) man page.

Cpuset System Man Pages

The `man` command provides online help on all resource management commands. To view a man page online, type `man` *commandname*.

User-Level Man Pages

The following user-level man pages are provided with Cpuset System software:

User-level man page	Description
`cpuset`(1)	Defines and manages a set of CPUs

Admin-Level Man Pages

The following system administrator-level man pages are provided with Cpuset System software:

User-level man page	Description
`bootcpuset`(8)	Places the specified process IDs (PIDs) into a bootcpuset of a configured size
`bootcpuset.rc`(8)	An `init` script, that creates the bootcpuset.

Cpuset Library Man Pages

The following cpuset library man pages are provided with Cpuset System software:

Cpuset library man page	Description
`cpusetAllocQueueDef`(3x)	Allocates a `cpuset_QueueDef_t` structure
`cpusetAttach`(3x)	Attaches the current process to a cpuset
`cpusetAttachPID`(3x)	Attaches a specific process to a cpuset
`cpusetCreate`(3x)	Creates a cpuset
`cpusetDestroy`(3x)	Destroys a cpuset
`cpusetDetachAll`(3x)	Detaches all threads from a cpuset

`cpusetDetachPID`(3x)	Detaches a specific process from a cpuset
`cpusetFreeCPUList`(3x)	Releases memory used by a `cpuset_CPUList_t` structure
`cpusetFreeNameList`(3x)	Releases memory used by a `cpuset_NameList_t` structure
`cpusetFreePIDList`(3x)	Releases memory used by a `cpuset_PIDList_t` structure
`cpusetFreeProperties`(3x)	Releases memory used by a `cpuset_Properties_t` structure **Not implemented on Linux**
`cpusetFreeQueueDef`(3x)	Releases memory used by a `cpuset_QueueDef_t` structure
`cpusetGetCPUCount`(3x)	Obtains the number of CPUs configured on the system
`cpusetGetCPUList`(3x)	Gets the list of all CPUs assigned to a cpuset
`cpusetGetName`(3x)	Gets the name of the cpuset to which a process is attached
`cpusetGetNameList`(3x)	Gets a list of names for all defined cpusets
`cpusetGetPIDList`(3x)	Gets a list of all PIDs attached to a cpuset
`cpusetGetProperties`(3x)	Retrieves various properties associated with a cpuset **Not implemented on Linux**

For more information on the cpuset library man pages, see Appendix A, "Application Programming Interface for the Cpuset System" on page 69.

File Format Man Pages

The following file format description man pages are provided with Cpuset System software:

File Format man page	Description
`cpuset`(4)	Cpuset configuration files
`bootcpuset.conf`(5)	Defines the number of nodes in a bootcpuset

Miscellaneous Man Pages

The following miscellaneous man pages are provided with Cpuset System software:

Miscellaneous man page	Description
`cpuset`(5)	Overview of the Cpuset System

NUMA Tools

This chapter describes the `dlook`(1) and `dplace`(1) tools that you can use to improve the performance of processes running on your SGI nonuniform memory access (NUMA) machine. You can use `dlook`(1) to find out where in memory the operating system is placing your application's pages and how much system and user CPU time it is consuming. You can use the `dplace`(1) command to bind a related set of processes to specific CPUs or nodes to prevent process migration. This can improve the performance of your application since it increases the percentage of memory accesses that are local.

Note: The information in this chapter was moved to the *Linux Configuration and Operations Guide*.

Application Programming Interface for the Cpuset System

This appendix contains information about cpusets system programming.

This appendix contains the following sections:

Overview

The cpuset library provides interfaces that allow a programmer to create and destroy cpusets, retrieve information about existing cpusets, obtain information about the properties associated with existing cpusets, and to attach a process and all of its children to a cpuset.

The cpuset library requires that a permission file be defined for a cpuset that is created. The permissions file may be an empty file, since it is only the file permissions for the file that define access to the cpuset. When permissions need to be checked, the current permissions of the file are used. It is therefore possible to change access to a particular cpuset without having to tear it down and recreate it, simply by changing the access permissions. Read access allows a user to retrieve information about a cpuset and execute permission allows the user to attach a process to the cpuset.

The cpuset library is provided as a Dynamic Shared Object (DSO) library. The library file is `libcpuset.so`, and it is normally located in the directory `/usr/lib`. Users of the library must include the `cpuset.h` header file, which is located in `/usr/include`. The function interfaces provided in the cpuset library are declared as optional interfaces to allow for backward compatibility as new interfaces are added to the library.

The function interfaces within the cpuset library include the following:

Function interface	Description
`cpusetCreate`(3x)	Creates a cpuset
`cpusetAttach`(3x)	Attaches the current process to a cpuset
`cpusetAttachPID`(3x)	Attaches a specific process to a cpuset
`cpusetDetachAll`(3x)	Detaches all threads from a cpuset
`cpusetDetachPID`(3x)	Detaches a specific process from a cpuset
`cpusetDestroy`(3x)	Destroys a cpuset
`cpusetGetCPUCount`(3x)	Obtains the number of CPUs configured on the system
`cpusetGetCPUList`(3x)	Gets the list of all CPUs assigned to a cpuset
`cpusetGetName`(3x)	Gets the name of the cpuset to which a process is attached
`cpusetGetNameList`(3x)	Gets a list of names for all defined cpusets
`cpusetGetPIDList`(3x)	Gets a list of all PIDs attached to a cpuset
`cpusetGetProperties`(3x)	Retrieves various properties associated with a cpuset
`cpusetAllocQueueDef`(3x)	Allocates a `cpuset_QueueDef_t` structure
`cpusetFreeQueueDef`(3x)	Releases memory used by a `cpuset_QueueDef_t` structure
`cpusetFreeCPUList`(3x)	Releases memory used by a `cpuset_CPUList_t` structure
`cpusetFreeNameList`(3x)	Releases memory used by a `cpuset_NameList_t` structure
`cpusetFreePIDList`(3x)	Releases memory used by a `cpuset_PIDList_t` structure

`cpusetFreeProperties`(3x)	Releases memory used by a `cpuset_Properties_t` structure

Management Functions

This section contains the man pages for the following Cpuset System library management functions:

`cpusetCreate`(3x)	Creates a cpuset
`cpusetAttach`(3x)	Attaches the current process to a cpuset
`cpusetAttachPID`(3x)	Attaches a specific process to a cpuset
`cpusetDetachPID`(3x)	Detaches a specific process from a cpuset
`cpusetDetachAll`(3x)	Detaches all threads from a cpuset
`cpusetDestroy`(3x)	Destroys a cpuset

cpusetCreate(3x)

NAME

cpusetCreate - creates a cpuset

SYNOPSIS

```
#include <cpuset.h>
int cpusetCreate(char *qname, cpuset_QueueDef_t *qdef);
```

DESCRIPTION

The cpusetCreate function is used to create a cpuset queue. Only processes running root user ID are allowed to create cpuset queues.

The qname argument is the name that will be assigned to the new cpuset. The name of the cpuset must be a 3 to 8 character string. Queue names having 1 or 2 characters are reserved for use by the operating system.

The qdef argument is a pointer to a cpuset_QueueDef_t structure (defined in the cpuset.h include file) that defines the attributes of the queue to be created. The memory for cpuset_QueueDef_t is allocated using cpusetAllocQueueDef(3x) and it is released using cpusetFreeQueueDef(3x). The cpuset_QueueDef_t structure is defined as follows:

```
typedef struct {
                    int                flags;
                    char               *permfile;
                    cpuset_CPUList_t   *cpu;
                } cpuset_QueueDef_t;
```

The flags member is used to specify various control options for the cpuset queue. It is formed by applying the bitwise exclusive-OR operator to zero or more of the following values:

Note: For the currrent SGI ProPack for Linux release, the operating system disregards the setting of the flags member, and always acts as if CPUSET_MEMORY_LOCAL was specified.

CPUSET_CPU_EXCLUSIVE	Defines a cpuset to be restricted. Only threads attached to the cpuset queue (descendents of an attached thread inherit the attachment) may

	execute on the CPUs contained in the cpuset.
CPUSET_MEMORY_LOCAL	Threads assigned to the cpuset will attempt to assign memory only from nodes within the cpuset. Assignment of memory from outside the cpuset will occur only if no free memory is available from within the cpuset. No restrictions are made on memory assignment to threads running outside the cpuset.
CPUSET_MEMORY_EXCLUSIVE	Threads assigned to the cpuset will attempt to assign memory only from nodes within the cpuset. Assignment of memory from outside the cpuset will occur only if no free memory is available from within the cpuset. Threads not assigned to the cpuset will not use memory from within the cpuset unless no memory outside the cpuset is available. If, at the time a cpuset is created, memory is already assigned to threads that are already running, no attempt will be made to explicitly move this memory. If page migration is enabled, the pages will be migrated when the system detects that most references to the pages are nonlocal.
CPUSET_MEMORY_KERNEL_AVOID	The kernel should attempt to avoid allocating memory from nodes contained in this cpuset. If kernel memory requests cannot be satisfied from outside this cpuset, this option will be ignored and allocations will occur from within the cpuset. (This avoidance currently extends only to

keeping buffer cache away from the protected nodes.)

The `permfile` member is the name of the file that defines the access permissions for the cpuset queue. The file permissions of `filename` referenced by `permfile` define access to the cpuset. Every time permissions need to be checked, the current permissions of this file are used. Thus, it is possible to change the access to a particular cpuset without having to tear it down and re-create it, simply by changing the access permissions. Read access to the `permfile` allows a user to retrieve information about a cpuset, and execute permission allows the user to attach a process to the cpuset.

The `cpu` member is a pointer to a `cpuset_CPUList_t` structure. The memory for the `cpuset_CPUList_t` structure is allocated and released when the `cpuset_QueueDef_t` structure is allocated and released (see `cpusetAllocQueueDef`(3x)). The CPU IDs listed here are (in the terms of the `cpumemsets`(2) man page) application, not system, numbers. The `cpuset_CPUList_t` structure contains the list of CPUs assigned to the cpuset. The `cpuset_CPUList_t` structure (defined in the `cpuset.h` include file) is defined as follows:

```
typedef struct {
                    int    count;
                    int    *list;
           } cpuset_CPUList_t;
```

The `count` member defines the number of CPUs contained in the list.

The `list` member is a pointer to the list (an allocated array) of the CPU IDs. The memory for the `list` array is allocated and released when the `cpuset_CPUList_t` structure is allocated and released.

EXAMPLES

This example creates a cpuset queue that has access controlled by the file `/usr/tmp/mypermfile`; contains CPU IDs 4, 8, and 12; and is CPU exclusive and memory exclusive:

```
cpuset_QueueDef_t *qdef;
               char              *qname = "myqueue";

               /* Alloc queue def for 3 CPU IDs */
               qdef = cpusetAllocQueueDef(3);
               if (!qdef) {
```

```
    perror("cpusetAllocQueueDef");
    exit(1);
}

/* Define attributes of the cpuset */
qdef->flags = CPUSET_CPU_EXCLUSIVE
            | CPUSET_MEMORY_EXCLUSIVE;
qdef->permfile = "/usr/tmp/mypermfile";
qdef->cpu->count = 3;
qdef->cpu->list[0] = 4;
qdef->cpu->list[1] = 8;
qdef->cpu->list[2] = 12;

/* Request that the cpuset be created */
if (!cpusetCreate(qname, qdef)) {
    perror("cpusetCreate");
    exit(1);
}
cpusetFreeQueueDef(qdef);
```

NOTES

The `cpusetCreate` function is found in the `libcpuset.so` library and is loaded if the `-lcpuset` option is used with either the `cc`(1) or `ld`(1) command.

SEE ALSO

`cpuset`(1), `cpusetAllocQueueDef`(3x), `cpusetFreeQueueDef`(3x), and `cpuset`(5).

DIAGNOSTICS

If successful, the `cpusetCreate` function returns a value of 1. If the `cpusetCreate` function fails, it returns the value 0 and `errno` is set to indicate the error. The possible values for `errno` include those values set by `fopen`(3), `cpumemsets`(2), and the following:

`ENODEV` Request for CPU IDs that do not exist on the system.

cpusetAttach(3x)

NAME

cpusetAttach - attaches the current process to a cpuset

SYNOPSIS

```
#include <cpuset.h>
int cpusetAttach( char *qname);
```

DESCRIPTION

The cpusetAttach function is used to attach the current process to the cpuset identified by qname. Every cpuset queue has a file that defines access permissions to the queue. The execute permissions for that file will determine if a process owned by a specific user can attach a process to the cpuset queue.

The qname argument is the name of the cpuset to which the current process should be attached.

EXAMPLES

This example attaches the current process to a cpuset queue named mpi_set.

```
char *qname = "mpi_set";

              /* Attach to cpuset, if error - print error & exit */
              if (!cpusetAttach(qname)) {
                  perror("cpusetAttach");
                  exit(1);
              }
```

NOTES

The cpusetAttach function is found in the libcpuset.so library and is loaded if the -lcpuset option is used with either the cc(1) or ld(1) command.

SEE ALSO

cpuset(1), cpusetCreate(3x), and cpuset(5).

DIAGNOSTICS

If successful, the `cpusetAttach` function returns a value of 1. If the `cpusetAttach` function fails, it returns the value 0 and `errno` is set to indicate the error. The possible values for `errno` are the same as those used by `cpumemsets`(2).

cpusetAttachPID(3x)

NAME

cpusetAttachPID - attaches a specific process to a cpusett

SYNOPSIS

```
#include <cpuset.h>
int cpusetAttachPID(qname, pid);
char *qname;
pid_t pid;
```

DESCRIPTION

The cpusetAttachPID function is used to attach a specific process identified by its PID to the cpuset identified by qname. Every cpuset queue has a file that defines access permissions to the queue. The execute permissions for that file will determine if a process owned by a specific user can attach a process to the cpuset queue.

The qname argument is the name of the cpuset to which the specified process should be attached.

EXAMPLES

This example attaches the current process to a cpuset queue named mpi_set.

```
char *qname = "mpi_set";

/* Attach to cpuset, if error - print error & exit */
if (!cpusetAttachPID(qname, pid)) {
perror("cpusetAttachPID");
exit(1);               }
```

NOTES

The cpusetAttachPID function is found in the library libcpuset.so, and is loaded if the -l cpuset option is used with either the cc(1) or ld(1) commands.

SEE ALSO

cpuset(1), cpusetCreate(3x), cpusetDetachPID(3x), and cpuset(5).

DIAGNOSTICS

If successful, the `cpusetAttachPID` function returns a 1. If the `cpusetAttachPID` function fails, it returns the value 0 and `errno` is set to indicate the error. The possible values for `errno` are the same as those used by `cpumemsets`(2).

cpusetDetachPID(3x)

NAME

cpusetDetachPID - detaches a specific process from a cpusett

SYNOPSIS

```
#include <cpuset.h>
int cpusetDetachPID(qname, pid);
char *qname;
pid_t pid;
```

DESCRIPTION

The cpusetDetachPID function is used to detach a specific process identified by its PID to the cpuset identified by qname.

The qname argument is the name of the cpuset from which the specified process should be detached.

EXAMPLES

This example detaches the current process from a cpuset queue named mpi_set.

```
char *qname = "mpi_set";

/* Detach from cpuset, if error - print error & exit */
if (!cpusetDetachPID(qname, pid)) {
perror("cpusetDetachPID");
exit(1);               }
```

NOTES

The cpusetDetachPID function is found in the library libcpuset.so, and is loaded if the -l cpuset option is used with either the cc(1) or ld(1) commands.

SEE ALSO

cpuset(1), cpusetCreate(3x), cpusetAttachPID(3x), and cpuset(5).

DIAGNOSTICS

If successful, `cpusetDetachPID` returns a 1. If `cpusetAttachPID` fails, it returns the value 0 and `errno` is set to indicate the error. The possible values for `errno` are the same as those used by `cpumemsets`(2).

cpusetDetachAll(3x)

NAME

cpusetDetachAll - detaches all threads from a cpuset

SYNOPSIS

```
#include <cpuset.h>
int cpusetDetachAll(char *qname);
```

DESCRIPTION

The cpusetDetachAll function is used to detach all threads currently attached to the specified cpuset. Only a process running with root user ID can successfully execute cpusetDetachAll.

The qname argument is the name of the cpuset that the operation will be performed upon.

For the currrent SGI ProPack for Linux release, processes detached from their cpuset using cpusetDetachAll are assigned a CpuMemSet identical to that of the kernel (see cpumemsets(2)). By default this will allow execution on any CPU. If the kernel was booted with the cpumemset_minimal=1 kernel boot command line option, this will only allow execution on CPU 0. Subsequent CpuMemSet administrative actions can also affect the current setting of the kernel CpuMemSet.

EXAMPLES

This example detaches the current process from a cpuset queue named mpi_set.

```
char *qname = "mpi_set";

               /* Detach all members of cpuset, if error - print error & exit */
               if (!cpusetDetachAll(qname)) {
                   perror("cpusetDetachAll");
                   exit(1);
               }
```

NOTES

The cpusetDetachAll function is found in the libcpuset.so library and is loaded if the -lcpuset option is used with either the cc(1) or ld(1) command.

SEE ALSO

`cpuset`(1), `cpusetAttach`(3x), and `cpuset`(5).

DIAGNOSTICS

If successful, the `cpusetDetachAll` function returns a value of 1. If the `cpusetDetachAll` function fails, it returns the value 0 and `errno` is set to indicate the error. The possible values for `errno` are the same as those used by `cpumemsets`(2).

cpusetDestroy(3x)

NAME

cpusetDestroy - destroys a cpuset

SYNOPSIS

```
#include <cpuset.h>
int cpusetDestroy(char *qname);
```

DESCRIPTION

The cpusetDestroy function is used to destroy the specified cpuset. The qname argument is the name of the cpuset that will be destroyed. Only processes running with root user ID are allowed to destroy cpuset queues. Any process currently attached to a destroyed cpuset can continue executing and forking children on the same processors and allocating memory in the same nodes, but no new processes may explicitly attach to a destroyed cpuset, nor otherwise reference it.

EXAMPLES

This example destroys the cpuset queue named mpi_set.

```
char *qname = "mpi_set";

              /* Destroy, if error - print error & exit */
              if (!cpusetDestroy(qname)) {
                  perror("cpusetDestroy");
                  exit(1);
              }
```

NOTES

The cpusetDestroy function is found in the libcpuset.so library and is loaded if the -lcpuset option is used with either the cc(1) or ld(1) command.

SEE ALSO

cpuset(1), cpusetCreate(3x), and cpuset(5).

Retrieval Functions

This section contains the man pages for the following Cpuset System library retrieval functions:

`cpusetGetCPUCount`(3x)	Obtains the number of CPUs configured on the system
`cpusetGetCPUList`(3x)	Gets the list of all CPUs assigned to a cpuset
`cpusetGetName`(3x)	Gets the name of the cpuset to which a process is attached
`cpusetGetNameList`(3x)	Gets a list of names for all defined cpusets
`cpusetGetPIDList`(3x)	Gets a list of all PIDs attached to a cpuset
`cpusetGetProperties`(3x)	Retrieves various properties associated with a cpuset
`cpusetAllocQueueDef`(3x)	Allocates a `cpuset_QueueDef_t` structure

cpusetGetCPUCount(3x)

NAME

cpusetGetCPUCount - obtains the number of CPUs configured on the system

SYNOPSIS

```
#include <cpuset.h>
int cpusetGetCPUCount(void);
```

DESCRIPTION

The cpusetGetCPUCount function returns the number of CPUs that are configured on the system.

EXAMPLES

This example obtains the number of CPUs configured on the system and then prints out the result.

```
int ncpus;

if (!(ncpus = cpusetGetCPUCount())) {
perror("cpusetGetCPUCount");
exit(1);
}
printf("The systems is configured for %d CPUs\n",
        ncpus);
```

NOTES

The cpusetGetCPUCount function is found in the libcpuset.so library and is loaded if the -lcpuset option is used with either the cc(1) or ld(1) command.

SEE ALSO

cpuset(1) and cpuset(5).

DIAGNOSTICS

If successful, the cpusetGetCPUCount function returns a value greater than or equal to the value of 1. If the cpusetGetCPUCount function fails, it returns the value 0 and errno is set to indicate the error. The possible values for errno are the same as those used by cpumemsets(2) and the following:

ERANGE	Number of CPUs configured on the system is not a value greater than or equal to 1.

cpusetGetCPUList(3x)

NAME

cpusetGetCPUList - gets the list of all CPUs assigned to a cpuset

SYNOPSIS

```
#include <cpuset.h>
cpuset_CPUList_t *cpusetGetCPUList(char *qname);
```

DESCRIPTION

The cpusetGetCPUList function is used to obtain the list of the CPUs assigned to the specified cpuset. Only processes running with a user ID or group ID that has read access permissions on the permissions file can successfully execute this function. The qname argument is the name of the specified cpuset.

The function returns a pointer to a structure of type cpuset_CPUList_t (defined in the cpuset.h include file). The function cpusetGetCPUList allocates the memory for the structure and the user is responsible for freeing the memory using the cpusetFreeCPUList(3x) function. The cpuset_CPUList_t structure looks similar to this:

```
typedef struct {
                    int    count;
                    pid_t  *list;
                } cpuset_CPUList_t;
```

The count member is the number of CPU IDs in the list. The list member references the memory array that holds the list of CPU IDs. The memory for list is allocated when the cpuset_CPUList_t structure is allocated and it is released when the cpuset_CPUList_t structure is released. The CPU IDs listed here are (in the terms of the cpumemsets(2) man page) application, not system, numbers.

EXAMPLES

This example obtains the list of CPUs assigned to the cpuset mpi_set and prints out the CPU ID values.

```
char *qname = "mpi_set";
    cpuset_CPUList_t *cpus;

    /* Get the list of CPUs else print error & exit */
    if (!( cpus = cpusetGetCPUList(qname))) {
```

```
    perror("cpusetGetCPUList");
exit(1);
}
if (cpus->count == 0) {
    printf("CPUSET[%s] has 0 assigned CPUs\n",
            qname);
} else {
    int i;

    printf("CPUSET[%s] assigned CPUs:\n",
            qname);
    for (i = 0; i < cpuset->count; i++)
        printf("CPU_ID[%d]\n", cpuset->list[i]);
}
cpusetFreeCPUList(cpus);
```

NOTES

The `cpusetGetCPUList` function is found in the `libcpuset.so` library and is loaded if the `-lcpuset` option is used with either the `cc`(1) or `ld`(1) command.

SEE ALSO

`cpuset`(1), `cpusetFreeCPUList`(3x), and `cpuset`(5).

DIAGNOSTICS

If successful, the `cpusetGetCPUList` function returns a pointer to a `cpuset_CPUList_t` structure. If the `cpusetGetCPUList` function fails, it returns NULL and `errno` is set to indicate the error. The possible values for `errno` include those values as set by `cpumemsets`(2) and `sbrk`(2).

cpusetGetName(3x)

NAME

cpusetGetName - gets the name of the cpuset to which a process is attached

SYNOPSIS

```
#include <cpuset.h>
cpuset_NameList_t *cpusetGetName(pid_t pid);
```

DESCRIPTION

The cpusetGetName function is used to obtain the name of the cpuset to which the specified process has been attached. The *pid* argument specifies the process ID.

The function returns a pointer to a structure of type cpuset_NameList_t (defined in the cpuset.h include file). The cpusetGetName function allocates the memory for the structure and all of its associated data. The user is responsible for freeing the memory using the cpusetFreeNameList(3x) function. The cpuset_NameList_t structure is defined as follows:

```
typedef struct {
                  int    count;
                  char   **list;
                  int   *status;
              } cpuset_NameList_t;
```

The count member is the number of cpuset names in the list. In the case of cpusetGetName function, this member should only contain the values of 0 and 1.

The list member references the list of names.

The status member is a list of status flags that indicate the status of the corresponding cpuset name in list. The following flag values may be used:

CPUSET_QUEUE_NAME	Indicates that the corresponding name in list is the name of a cpuset queue
CPUSET_CPU_NAME	Indicates that the corresponding name in list is the CPU ID for a restricted CPU

The memory for list and status is allocated when the cpuset_NameList_t structure is allocated and it is released when the cpuset_NameList_t structure is released.

EXAMPLES

This example obtains the cpuset name or CPU ID to which the current process is attached:

```
cpuset_NameList_t *name;

              /* Get the list of names else print error & exit */
              if (!(name = cpusetGetName(0))) {
                  perror("cpusetGetName");
                  exit(1);
              }
              if (name->count == 0) {
                  printf("Current process not attached\n");
              } else {
                  if (name->status[0] == CPUSET_CPU_NAME) {
                      printf("Current process attached to"
                              " CPU_ID[%s]\n",
                              name->list[0]);
                  } else {
                      printf("Current process attached to"
                              " CPUSET[%s]\n",
                              name->list[0]);
                  }
              }
              cpusetFreeNameList(name);
```

NOTES

The `cpusetGetName` function is found in the `libcpuset.so` library and is loaded if the `-lcpuset` option is used with either the `cc`(1) or `ld`(1) command.

This operation is not atomic and if multiple cpusets are defined with exactly the same member CPUs, not a recommended configuration, this call will return the first matching cpuset.

Restricted CPUs are not supported in the current SGI ProPack for Linux release.

SEE ALSO

`cpuset`(1), `cpusetFreeNameList`(3x), `cpusetGetNameList`(3x), and `cpuset`(5).

DIAGNOSTICS

If successful, the `cpusetGetName` function returns a pointer to a `cpuset_NameList_t` structure. If the `cpusetGetName` function fails, it returns NULL and `errno` is set to indicate the error. The possible values for `errno` include those values as set by `cpumemsets`(2), `sbrk`(2), and the following:

`EINVAL`	Invalid value for *pid* was supplied. Currently, only 0 is accepted to obtain the cpuset name that the current process is attached to.
`ERANGE`	Number of CPUs configured on the system is not a value greater than or equal to 1.

cpusetGetNameList(3x)

NAME

cpusetGetNameList - gets the list of names for all defined cpusets

SYNOPSIS

```
#include <cpuset.h>
cpuset_NameList_t *cpusetGetNameList(void);
```

DESCRIPTION

The cpusetGetNameList function is used to obtain a list of the names for all the cpusets on the system.

The cpusetGetNameList function returns a pointer to a structure of type cpuset_NameList_t (defined in the cpuset.h include file). The cpusetGetNameList function allocates the memory for the structure and all of its associated data. The user is responsible for freeing the memory using the cpusetFreeNameList(3x) function. The cpuset_NameList_t structure is defined as follows:

```
typedef struct {
                    int    count;
                    char   **list;
                    int   *status;
                } cpuset_NameList_t;
```

The count member is the number of cpuset names in the list.

The list member references the list of names.

The status member is a list of status flags that indicate the status of the corresponding cpuset name in list. The following flag values may be used:

CPUSET_QUEUE_NAME	Indicates that the corresponding name in list is the name of a cpuset queue.
CPUSET_CPU_NAME	Indicates that the corresponding name in list is the CPU ID for a restricted CPU.

The memory for list and status is allocated when the cpuset_NameList_t structure is allocated and it is released when the cpuset_NameList_t structure is released.

EXAMPLES

This example obtains the list of names for all cpuset queues configured on the system. The list of cpusets or restricted CPU IDs is then printed.

```
cpuset_NameList_t *names;

/* Get the list of names else print error & exit */
if (!(names = cpusetGetNameList())) {
    perror("cpusetGetNameList");
    exit(1);
}
if (names->count == 0) {
    printf("No defined CPUSETs or restricted CPUs\n");
} else {
    int i;

    printf("CPUSET and restricted CPU names:\n");
    for (i = 0; i < names->count; i++) {
        if (names->status[i] == CPUSET_CPU_NAME) {
            printf("CPU_ID[%s]\n", names->list[i]);
        } else {
            printf("CPUSET[%s]\n", names->list[i]);
        }
    }
}
cpusetFreeNameList(names);
```

NOTES

The `cpusetGetNameList` function is found in the `libcpuset.so` library and is loaded if the `-lcpuset` option is used with either the `cc`(1) or `ld`(1) command.

Restricted CPUs are not supported in the current SGI ProPack for Linux release.

SEE ALSO

`cpuset`(1), `cpusetFreeNameList`(3x), and `cpuset`(5).

DIAGNOSTICS

If successful, the `cpusetGetNameList` function returns a pointer to a `cpuset_NameList_t` structure. If the `cpusetGetNameList` function fails, it

returns NULL and `errno` is set to indicate the error. The possible values for `errno` include those values set by `cpumemsets`(2) and `sbrk`(2).

cpusetGetPIDList(3x)

NAME

cpusetGetPIDList - gets a list of all PIDs attached to a cpuset

SYNOPSIS

```
#include <cpuset.h>
cpuset_PIDList_t *cpusetGetPIDList(char *qname);
```

DESCRIPTION

The cpusetGetPIDList function is used to obtain a list of the PIDs for all processes currently attached to the specified cpuset. Only processes with a user ID or group ID that has read permissions on the permissions file can successfully execute this function.

The qname argument is the name of the cpuset to which the current process should be attached.

The function returns a pointer to a structure of type cpuset_PIDList_t (defined in the cpuset.h) include file. The cpusetGetPIDList function allocates the memory for the structure and the user is responsible for freeing the memory using the cpusetFreePIDList(3x) function. The cpuset_PIDList_t structure looks similar to this:

```
typedef struct {
                    int    count;
                    pid_t  *list;
                } cpuset_PIDList_t;
```

The count member is the number of PID values in the list. The list member references the memory array that holds the list of PID values. The memory for list is allocated when the cpuset_PIDList_t structure is allocated and it is released when the cpuset_PIDList_t structure is released.

EXAMPLES

This example obtains the list of PIDs attached to the cpuset mpi_set and prints out the PID values.

```
(char             *qname = "mpi_set";)
cpuset_PIDList_t *pids;
```

```
/* Get the list of PIDs else print error & exit */
if (!(pids = cpusetGetPIDList(qname))) {
    perror("cpusetGetPIDList");
    exit(1);
}
if (pids->count == 0) {
    printf("CPUSET[%s] has 0 processes attached\n",
            qname);
} else {
    int i;
    printf("CPUSET[%s] attached PIDs:\n",
             qname);
    for (i=o; i<pids->count; i++)
             printf("PID[%d]\n", pids->list[i] );
}
cpusetFreePIDList(pids);
```

NOTES

The `cpusetGetPIDList` function is found in the `libcpuset.so` library and is loaded if the `-lcpuset` option is used with either the `cc`(1) or `ld`(1) command.

This function scans the `/proc` table to determine cpuset membership and is therefore not atomic and the results cannot be guaranteed on a rapidly changing system.

SEE ALSO

`cpuset`(1), `cpusetFreePIDList`(3x), and `cpuset`(5).

DIAGNOSTICS

If successful, the `cpusetGetPIDList` function returns a pointer to a `cpuset_PIDList_t` structure. If the `cpusetGetPIDList` function fails, it returns NULL and `errno` is set to indicate the error. The possible values for `errno` are the same as the values set by `cpumemsets`(2) and `sbrk`(2).

cpusetGetProperties(3x)

NAME

cpusetGetProperties - retrieves various properties associated with a cpuset

SYNOPSIS

```
#include <cpuset.h>
cpuset_Properties_t * cpusetGetProperties(char *qname);
```

DESCRIPTION

The cpusetGetProperties function is used to retrieve various properties identified by qname and returns a pointer to a cpuset_Properties_t structure shown in the following:

```
/* structure to return cpuset properties */
typedef struct {
        cpuset_CPUList_t        *cpuInfo; /* cpu count and list */
        int                     pidCnt;   /* number of process in cpuset */
        uid_t                   owner;    /* owner id of config file */
        gid_t                   group;    /* group id of config file */
        mode_t                  DAC;      /* Standard permissions of
config file*/
        int                     flags;    /* Config file flags for cpuset */
        int                     extFlags; /* Bit flags indicating valid
ACL & MAC */
        struct acl              accAcl;   /* structure for valid access
ACL */
        struct acl              defAcl;   /* structure for valid default
ACL */
        mac_label               macLabel; /* structure for valid MAC
label */
} cpuset_Properties_t;
```

Every cpuset queue has a file that defines access permissions to the queue. The read permissions for that file will determine if a process owned by a specific user can retrieve the properties from the cpuset.

The qname argument is the name of the cpuset to which the properties should be retrieved.

EXAMPLES

This example retrieves the properties of a cpuset queue named `mpi_set`.

```
char *qname = "mpi_set";
                  cpuset_Properties_t             *csp;


                  /* Get properties, if error - print error & exit */
                  csp=cpusetGetProperties(qname);
                  if (!csp) {
                          perror("cpusetGetProperties");
                          exit(1);
                  }
                  .
                  .
                  .
                  cpusetFreeProperties(csp);
```

Once a valid pointer is returned, a check against the `extFlags` member of the `cpuset_Properties_t` structure must be made with the flags `CPUSET_ACCESS_ACL`, `CPUSET_DEFAULT_ACL`, and `CPUSET_MAC_LABEL` to see if any valid ACLs or a valid MAC label was returned. The check flags can be found in the `sn\cpuset.h` file.

NOTES

The `cpusetGetProperties` function is found in the `libcpuset.so` library and is loaded if the `-lcpuset` option is used with either the `cc`(1) or `ld`(1) command.

Access control lists (ACLs) and mandatory access lists (MACs) are not implemented in the current SGI ProPack for Linux release.

SEE ALSO

`cpuset`(1), `cpusetFreeProperties`(3x), and `cpuset`(5).

DIAGNOSTICS

If successful, the `cpusetGetProperties` function returns a pointer to a `cpuset_Properties_t` structure. If the `cpusetGetProperties` function fails, it returns NULL and `errno` is set to indicate the error. The possible values for `errno` include those values set by `cpumemsets`(2).

cpusetAllocQueueDef(3x)

NAME

cpusetAllocQueueDef - allocates a cpuset_QueueDef_t structure

SYNOPSIS

```
#include <cpuset.h>
cpuset_QueueDef_t *cpusetAllocQueueDef(int count)
```

DESCRIPTION

The cpusetAllocQueueDef function is used to allocate memory for a cpuset_QueueDef_t structure. This memory can then be released using the cpusetFreeQueueDef(3x) function.

The count argument indicates the number of CPUs that will be assigned to the cpuset definition structure. The cpuset_QueueDef_t structure is defined as follows:

```
typedef struct {
                    int                flags;
                    char               *permfile;
                    cpuset_CPUList_t   *cpu;
                } cpuset_QueueDef_t;
```

The flags member is used to specify various control options for the cpuset queue. It is formed by applying the bitwise exclusive-OR operator to zero or more of the following values:

Note: For the currrent SGI ProPack for Linux release, the operating system disregards the setting of the flags member, and always acts as if CPUSET_MEMORY_LOCAL was specified.

CPUSET_CPU_EXCLUSIVE	Defines a cpuset to be restricted. Only threads attached to the cpuset queue (descendents of an attached thread inherit the attachement) may execute on the CPUs contained in the cpuset.
CPUSET_MEMORY_LOCAL	Threads assigned to the cpuset will attempt to assign memory only from nodes within the cpuset.

	Assignment of memory from outside the cpuset will occur only if no free memory is available from within the cpuset. No restrictions are made on memory assignment to threads running outside the cpuset.
`CPUSET_MEMORY_EXCLUSIVE`	Threads assigned to the cpuset will attempt to assign memory only from nodes within the cpuset. Assignment of memory from outside the cpuset will occur only if no free memory is available from within the cpuset. Threads not assigned to the cpuset will not use memory from within the cpuset unless no memory outside the cpuset is available. If, at the time a cpuset is created, memory is already assigned to threads that are already running, no attempt will be made to explicitly move this memory. If page migration is enabled, the pages will be migrated when the system detects that most references to the pages are nonlocal.
`CPUSET_MEMORY_KERNEL_AVOID`	The kernel should attempt to avoid allocating memory from nodes contained in this cpuset. If kernel memory requests cannot be satisfied from outside this cpuset, this option will be ignored and allocations will occur from within the cpuset. (This avoidance currently extends only to keeping buffer cache away from the protected nodes.)

The `permfile` member is the name of the file that defines the access permissions for the `cpuset` queue. The file permissions of `filename` referenced by `permfile` define access to the cpuset. Every time permissions need to be checked, the current permissions of this file are used. Thus, it is possible to change the access to a

particular cpuset without having to tear it down and re-create it, simply by changing the access permissions. Read access to the `permfile` allows a user to retrieve information about a cpuset, and execute permission allows the user to attach a process to the cpuset.

The `cpu` member is a pointer to a `cpuset_CPUList_t` structure. The memory for the `cpuset_CPUList_t` structure is allocated and released when the `cpuset_QueueDef_t` structure is allocated and released (see `cpusetFreeQueueDef`(3x)). The `cpuset_CPUList_t` structure contains the list of CPUs assigned to the cpuset. The `cpuset_CPUList_t` structure (defind in the `cpuset.h` include file) is defined as follows:

```
typedef struct {
                    int    count;
                    int    *list;
                } cpuset_CPUList_t;
```

The `count` member defines the number of CPUs contained in the list.

The `list` member is the pointer to the list (an allocated array) of the CPU IDs. The memory for the list array is allocated and released when the `cpuset_CPUList_t` structure is allocated and released. The size of the list is determined by the `count` argument passed into the function `cpusetAllocQueueDef`. The CPU IDs listed here are (in the terms of the `cpumemsets`(2) man page) application, not system, numbers.

EXAMPLES

This example creates a cpuset queue using the `cpusetCreate`(3x) function and provides an example of how the `cpusetAllocQueueDef` function might be used. The cpuset created will have access controlled by the file `/usr/tmp/mypermfile`; it will contain CPU IDs 4, 8, and 12; and it will be CPU exclusive and memory exclusive:

```
cpuset_QueueDef_t *qdef;
                char              *qname = "myqueue";

                /* Alloc queue def for 3 CPU IDs */
                qdef = cpusetAllocQueueDef(3);
                if (!qdef) {
                    perror("cpusetAllocQueueDef");
                    exit(1);
                }

                /* Define attributes of the cpuset */
```

```
qdef->flags = CPUSET_CPU_EXCLUSIVE
            | CPUSET_MEMORY_EXCLUSIVE;
qdef->permfile = "/usr/tmp/mypermfile";
qdef->cpu->count = 3;
qdef->cpu->list[0] = 4;
qdef->cpu->list[1] = 8;
qdef->cpu->list[2] = 12;

/* Request that the cpuset be created */
if (!cpusetCreate(qname, qdef)) {
    perror("cpusetCreate");
    exit(1);
}
cpusetFreeQueueDef(qdef);
```

NOTES

The `cpusetAllocQueueDef` function is found in the `libcpuset.so` library and is loaded if the `-lcpuset` option is used with either the `cc`(1) or `ld`(1) command.

The current SGI ProPack for Linux release disregards the setting of the `flags` member and always acts as if `CPUSET_MEMORY_LOCAL` was specified.

SEE ALSO

`cpuset`(1), `cpusetFreeQueueDef`(3x), and `cpuset`(5).

DIAGNOSTICS

If successful, the `cpusetAllocQueueDef` function returns a pointer to a `cpuset_QueueDef_t` structure. If the `cpusetAllocQueueDef` function fails, it returns NULL and `errno` is set to indicate the error. The possible values for `errno` values include those returned by `sbrk`(2) and the following:

`EINVAL`	Invalid argument was supplied. The user must supply a value greater than or equal to 0.

Clean-up Functions

This section contains the man pages for Cpuset System library clean-up functions:

`cpusetFreeQueueDef`(3x)	Releases memory used by a `cpuset_QueueDef_t` structure
`cpusetFreeCPUList`(3x)	Releases memory used by a `cpuset_CPUList_t` structure
`cpusetFreeNameList`(3x)	Releases memory used by a `cpuset_NameList_t` structure
`cpusetFreePIDList`(3x)	Releases memory used by a `cpuset_PIDList_t` structure
`cpusetFreeProperties`(3x)	Release memory used by a `cpuset_Properties_t` structure

cpusetFreeQueueDef(3x)

NAME

cpusetFreeQueueDef - releases memory used by a cpuset_QueueDef_t structure

SYNOPSIS

```
#include <cpuset.h>
void cpusetFreeQueueDef(cpuset_QueueDef_t *qdef);
```

DESCRIPTION

The cpusetFreeQueueDef function is used to release memory used by a cpuset_QueueDef_t structure. This function releases all memory associated with the cpuset_QueueDef_t structure.

The qdef argument is the pointer to the cpuset_QueueDef_t structure that will have its memory released.

This function should be used to release the memory allocated during a previous call to the cpusetAllocQueueDef(3x)) function.

NOTES

The cpusetFreeQueueDef function is found in the libcpuset.so library and is loaded if the -lcpuset option is used with either the cc(1) or ld(1) command.

SEE ALSO

cpuset(1), cpusetAllocQueueDef(3x), and cpuset(5).

cpusetFreeCPUList(3x)

NAME

cpusetFreeCPUList - releases memory used by a cpuset_CPUList_t structure

SYNOPSIS

```
#include <cpuset.h>
void cpusetFreeCPUList(cpuset_CPUList_t *cpu);
```

DESCRIPTION

The cpusetFreeCPUList function is used to release memory used by a cpuset_CPUList_t structure. This function releases all memory associated with the cpuset_CPUList_t structure.

The cpu argument is the pointer to the cpuset_CPUList_t structure that will have its memory released.

This function should be used to release the memory allocated during a previous call to the cpusetGetCPUList(3x) function.

NOTES

The cpusetFreeCPUList function is found in the libcpuset.so library and is loaded if the -lcpuset option is used with either the cc(1) or ld(1) command.

SEE ALSO

cpuset(1), cpusetGetCPUList(3x), and cpuset(5).

cpusetFreeNameList(3x)

NAME

cpusetFreeNameList - releases memory used by a cpuset_NameList_t structure

SYNOPSIS

```
#include <cpuset.h>
void cpusetFreeNameList(cpuset_NameList_t *name);
```

DESCRIPTION

The cpusetFreeNameList function is used to release memory used by a cpuset_NameList_t structure. This function releases all memory associated with the cpuset_NameList_t structure.

The name argument is the pointer to the cpuset_NameList_t structure that will have its memory released.

This function should be used to release the memory allocated during a previous call to the cpusetGetNameList(3x) function or cpusetGetName(3x) function.

NOTES

The cpusetFreeNameList function is found in the libcpuset.so library and is loaded if the -lcpuset option is used with either the cc(1) or ld(1) command.

SEE ALSO

cpuset(1), cpusetGetName(3x), cpusetGetNameList(3x), and cpuset(5).

cpusetFreePIDList(3x)

NAME

cpusetFreePIDList - releases memory used by a cpuset_PIDList_t structure

SYNOPSIS

```
#include <cpuset.h>
void cpusetFreePIDList(cpuset_PIDList_t *pid);
```

DESCRIPTION

The cpusetFreePIDList function is used to release memory used by a cpuset_PIDList_t structure. This function releases all memory associated with the cpuset_PIDList_t structure.

The pid argument is the pointer to the cpuset_PIDList_t structure that will have its memory released.

This function should be used to release the memory allocated during a previous call to the cpusetGetPIDList(3x) function.

NOTES

The cpusetFreePIDList function is found in the libcpuset.so library and is loaded if the -lcpuset option is used with either the cc(1) or ld(1) command.

SEE ALSO

cpuset(1), cpusetGetPIDList(3x), and cpuset(5).

cpusetFreeProperties(3x)

NAME

cpusetFreeProperties - releases memory used by a cpuset_Properties_t structure

SYNOPSIS

```
#include <cpuset.h>
void cpusetFreeProperties(cpuset_Properties_t *csp);
```

DESCRIPTION

The cpusetFreeProperties function is used to release memory used by a cpuset_Properties_t structure. This function releases all memory associated with the cpuset_Properties_t structure.

The csp argument is the pointer to the cpuset_Properties_t structure that will have its memory released.

This function should be used to release the memory allocated during a previous call to the cpusetGetProperties(3x)) function.

NOTES

The cpusetFreeProperties function is found in the libcpuset.so library and is loaded if the -lcpuset option is used with either the cc(1) or ld(1) command.

SEE ALSO

cpuset(1), cpusetGetProperties(3x), and cpuset(5).

Using the Cpuset Library

This section provides an example of how to use the Cpuset library functions to create a cpuset and an example of creating a replacement library for `/lib32/libcpuset.so`.

Example A-1 Example of Creating a Cpuset

This example creates a cpuset named `myqueue` containing CPUs 4, 8, and 12. The example uses the interfaces in the cpuset library, `/usr/lib/libcpuset.so`, if they are present.

```
#include <cpuset.h>
#include <stdio.h>
#include <errno.h>

#define PERMFILE "/usr/tmp/permfile"

int
main(int argc, char **argv)
{
    cpuset_QueueDef_t *qdef;
    char              *qname = "myqueue";
    FILE              *fp;

    /* Alloc queue def for 3 CPU IDs */
    if (_MIPS_SYMBOL_PRESENT(cpusetAllocQueueDef)) {
        printf("Creating cpuset definition\n");
        qdef = cpusetAllocQueueDef(3);
        if (!qdef) {
            perror("cpusetAllocQueueDef");
            exit(1);
    }
        /* Define attributes of the cpuset */
        qdef->flags = CPUSET_CPU_EXCLUSIVE
                    | CPUSET_MEMORY_LOCAL
                    | CPUSET_MEMORY_EXCLUSIVE;
        qdef->permfile = PERMFILE;
        qdef->cpu->count = 3;
        qdef->cpu->list[0] = 4;
        qdef->cpu->list[1] = 8;
        qdef->cpu->list[2] = 12;
    } else {
```

```
    printf("Writing cpuset command config"
            " info into %s\n", PERMFILE);
    fp = fopen(PERMFILE, "a");
    if (!fp) {
        perror("fopen");
        exit(1);
    }
    fprintf(fp, "EXCLUSIVE\n");
    fprintf(fp, "MEMORY_LOCAL\n");
    fprintf(fp, "MEMORY_EXCLUSIVE\n\n");
    fprintf(fp, "CPU 4\n");
    fprintf(fp, "CPU 8\n");
    fprintf(fp, "CPU 12\n");
    fclose(fp);
}

/* Request that the cpuset be created */
if (_MIPS_SYMBOL_PRESENT(cpusetCreate)) {
    printf("Creating cpuset = %s\n", qname);
    if (!cpusetCreate(qname, qdef)) {
        perror("cpusetCreate");
        exit(1);
    }
} else {
    char command[256];

    fprintf(command, "/usr/sbin/cpuset -q %s -c"
           "-f %s", qname,
           [PERMFILE];
    if (system(command) < 0) {
        perror("system");
        exit(1);
    }
}

/* Free memory for queue def */
if (_MIPS_SYMBOL_PRESENT(cpusetFreeQueueDef)) {
    printf("Finished with cpuset definition,"
            " releasing memory\n");
    cpusetFreeQueueDef(qdef);
    }
```

```
    return 0;
}
```

Index

J

L

M