Linux® Resource Administration Guide

007–4413–007

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 SGI ProPack 3 for Linux Service Pack 4 and the SGI ProPack 4 for Linux operating systems.

Major Documentation Changes

Added information about the `jstat`(1), `jwait`(1), and `jkill`(1) commands to "Linux Kernel Job Overview" on page 1.

Added information about the job library in "Job Library" on page 3.

Added information about user account information for SGI ProPack 4 for Linux in "Installing and Configuring Array Services".

Added a new chapter describing cpusets on an SGI ProPack 4 for Linux system in Chapter 6, "Cpusets on SGI ProPack 4 for Linux" on page 121.

Added a new appendix describing the SGI ProPack 4 for Linux `libcpuset` C programming application programming interface (API) functions in Appendix B, "SGI ProPack 4 Cpuset Library Functions" on page 189.

Moved the Comprehensive System Accounting application programming interface information into Appendix C, "Application Programming Interface for the Comprehensive System Accounting (CSA)" on page 215.

Record of Revision

Version	Description
001	February 2003 Original publication.
002	June 2003 Updated to support the SGI ProPack for Linux v2.2 release
003	October 2003 Updated to support the SGI ProPack for Linux v2.3 release.
004	February 2004 Updated to support the SGI ProPack for Linux v2.4 release.
005	May 2004 Updated to support the SGI ProPack 3 for Linux release.
006	January 2005 Updated to support the SGI ProPack 3 for Linux Service Pack 3 release.
007	February 2005 Updated to support the SGI ProPack 4 for Linux release.

Contents

Figures

Tables

About This Guide

This guide is a reference document for people who manage the operation of SGI computer systems running either the SGI ProPack 3 for Linux or SGI ProPack 4 for Linux operating system. It contains information needed in the administration of various system resource management features.

This manual contains the following chapters:

Related Publications

For a list of manuals supporting SGI Linux releases, see the *SGI ProPack for Linux Start Here* or the *SGI ProPack 4 for Linux Start Here*, respectively.

For a list of Array Services man pages, see "Using Array Services Commands" on page 64.

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.
- The information in this guide, other SGI ProPack 3 for Linux documentation, and all other documentation included in the RPMs on the distribution CDs can be found on the CD titled "SGI ProPack 3 for Linux Service Pack 3 - 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.

  Note: The release notes, which contain the latest information about software and documentation in this release, are on the SGI ProPack 3 for Linux Service Pack 3 Documentation CD (CD4) in the root directory, in a file named `README.TXT`.

- Online versions of the *SGI ProPack 4 for Linux Start Here*, the SGI ProPack 4 release notes, which contain the latest information about software and documentation in this release, the list of RPMs distributed with SGI ProPack 4, and a useful migration guide, which contains helpful hints and advice for customers moving from earlier versions of SGI ProPack to SGI ProPack 4, can be found in the `/docs` directory on the SGI ProPack 4 Open/Free Source CD.

  The SGI ProPack 4 for Linux release notes get installed to the following location on a system running SGI ProPack 4:
  `/usr/share/doc/sgi-propack-4/README.txt`
- 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:

Linux Kernel Job 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.

Job initiators can be categorized as either interactive or batch processes.

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.

The job container can be used stand-alone. The batch scheduler LSF, for example, uses jobs directly to keep track of the collection of processes that make up a single batch scheduler request.

Figure 1-1 on page 2, shows the point-of-entry processes that initiate the creation of jobs.

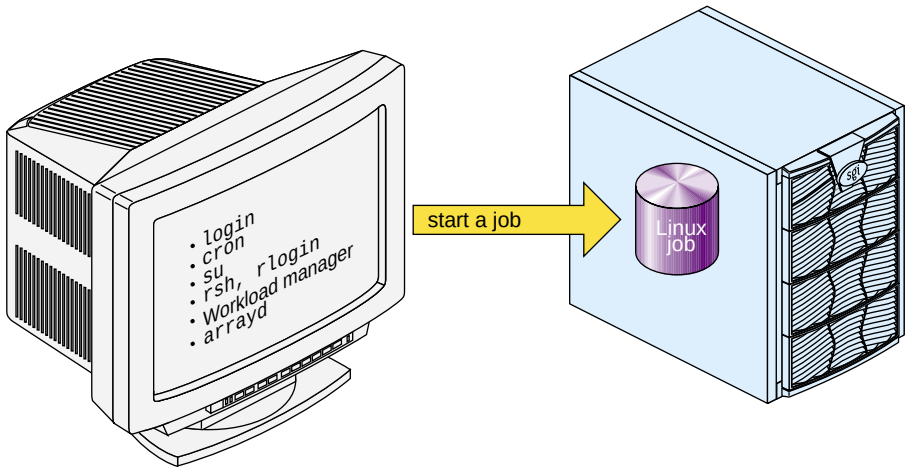

Figure 1-1 Point-of-Entry Processes

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.

For more information on jobs, see `job`(7) man page.

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.

You can use the `jstat`(1) command to display job status information. The `jwait`(1) command waits for the job whose job ID is defined by the `jid` parameter and reports its termination status. The termination status is determined based upon the last process to exit the job. The root user can wait on any process on the system. All other users can only wait on jobs that they own. The `jkill`(1) command sends the specified signal to the processes contained in the job(s) identified by the `jid`(s). If no signal is specified, the TERM signal is sent. The TERM signal will kill processes which do not catch this signal. For other processes, if may be necessary to use the `KILL` (9) signal, since this signal cannot be caught. For more information on these job commands, see the appropriate man page.

Job Library

The `job_create` library call is part of the job library that allows processes to manipulate and obtain status about Linux jobs. When the job library is to be used, the `job.h` header file should be included to obtain the proper definitions.

The syntax of the `job_create` call is, as follows:

```
#include <job.h>

jid_t job_create( jid_t jid_requested, uid_t uid, int options );
```

The `job_create` call creates a new job and attaches the calling process to the new job.

The `jid_requested` parameter allows the caller to specify a`jid` value to use when creating the job. If the caller wants the system to generate the job ID, then set `jid_requested` to 0. The `uid` parameter is used to supply the user ID value for the user that will own the job. For more information, see the `job_create`(3) man page.

The following routines are part of the job library:

Library Routine	Description
`job_detachjid`(3)	Detaches all the processes from a job

`job_detachpid`(3)	Detaches a process from its current job
`job_getjid`(3)	Returns the job ID for the given process
`job_getjidcnt`(3)	Returns the number of jobs currently on the system
`job_getjidlist`(3)	Gets the `jid`s of the currently active job
`job_getpidcnt`(3)	Gets the number of processes attached to a job
`job_getpidlist`(3)	Gets the list of process `pid`s attached to a job
`job_getprimepid`(3)	Gets the prime process `pid` for a job
`job_getuid`(3)	Gets the user ID of a job
`job_killjid`(3)	Sends a kill signal to all processes in a job
`job_sethid`(3)	Allows processes to manipulate and obtain status about Linux jobs.
`job_waitjid`(3)	Waits for a job to complete

For more information about these job library routines, see the appropriate man page.

Installing and Configuring Linux Kernel Jobs for Use with CSA

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
```

Comprehensive System Accounting

Comprehensive System Accounting (CSA) provides detailed, accurate accounting data per job. It also provides data from some daemons. CSA typically runs with Linux kernel job. For more information on Linux kernel jobs, see Chapter 1, "Linux Kernel Jobs" on page 1. If you run CSA without Linux kernel jobs installed, no job accounting would be available.

The `csarun`(8) command, usually initiated by the `cron`(8) command, directs the processing of the CSA daily accounting files. The `csarun`(8) command processes accounting records written into the CSA accounting data file.

Using accounting data, you can determine how system resources were used and if a particular user has used more than a reasonable share; trace significant system events, such as security breaches, by examining the list of all processes invoked by a particular user at a particular time; and set up billing systems to charge login accounts for using system resources.

The Linux CSA application interface library allows software applications to manipulate and obtain status about Linux CSA accounting methods. For more information, see " Linux CSA Application Interface Library" on page 56 and "Linux CSA Application Interface Library" on page 215.

This chapter contains the following sections:

CSA Overview

Comprehensive System Accounting (CSA) is a set of C programs and shell scripts that, like the other accounting packages, provide methods for collecting per-process resource usage data, monitoring disk usage, and charging fees to specific login accounts. CSA provides:

- Per-job accounting
- Daemon accounting (workload management systems and tape systems; note that tape daemon accounting is not supported in this release)
- Flexible accounting periods (daily and periodic (monthly) accounting reports can be generated as often as desired and are not restricted to once per day or once per month)
- Flexible system billing units (SBUs)
- Offline archiving of accounting data
- User exits for site specific customizing of daily and periodic (monthly) accounting
- Configurable parameters within the `/etc/csa.conf` file
- User job accounting (`ja`(1) command)

CSA takes this per-process accounting information and combines it by job identifier (`jid`) within system boot uptime periods. CSA accounting for a job consists of all accounting data for a given job identifier during a single system boot period. However, since workload management jobs may span multiple reboots and thereby consist of multiple job identifiers, CSA accounting for these jobs includes the accounting data associated with the workload management identifier. For this release, the workload managment identifier is yet to be defined.

Daemon accounting records are written at the completion of daemon specific events. These records are combined with per-process accounting records associated with the same job.

By default, CSA only reports accounting data for terminated jobs. Interactive jobs, `cron` jobs and `at` jobs terminate when the last process in the job exits, which is normally the login shell. A workload management job is recognized as terminated by CSA based upon daemon accounting records and an end-of-job record for that job. Jobs which are still active are recycled into the next accounting period. This behavior can be changed through use of the `csarun` command `-A` option.

A system billing unit (SBU) is a unit of measure that reflects use of machine resources. SBUs are defined in the CSA configuration file `/etc/csa.conf` and are set to `0.0` by default. The weighting factor associated with each field in the CSA accounting records can be altered to obtain an SBU value suitable for your site. For more information on SBUs, see "System Billing Units (SBUs)" on page 40.

The CSA accounting records are written into a separate CSA `/var/csa/day/pacct` file. The CSA commands can only be used with CSA generated accounting records.

There are four user exits available with the `csarun`(8) daily accounting script. There is one user exit available with the `csaperiod`(8) monthly accounting script. These user exits allow sites to tailor the daily and monthly run of accounting to their specific needs by creating user exit scripts to perform any additional processing and to allow archiving of accounting data. See the `csarun`(8) and `csaperiod`(8) man pages for further information. (User exits have not been defined for this release).

CSA provides two user accounting commands, `csacom`(1) and `ja`(1). The `csacom` command reads the CSA `pacct` file and writes selected accounting records to standard output. The `ja` command provides job accounting information for the current job of the caller. This information is obtained from a separate user job accounting file to which the kernel writes. See the `csacom`(1) and `ja`(1) man pages for further information.

"Workload Management Requests and Recycled Data" on page 39, contains information on cleaning up and maintaining workload management data files.

The `/etc/csa.conf` file contains CSA configuration variables. These variables are used by the CSA commands.

CSA is disabled in the kernel by default. To enable CSA, see "Enabling or Disabling CSA" on page 11.

Concepts and Terminology

The following concepts and terms are important to understand when using the accounting feature:

Term	Description
Daily accounting	Daily accounting is the processing, organizing, and reporting of the raw accounting data, generally performed once per day. In CSA, daily accounting can be run as many times as necessary during a day; however, this feature is still referred to as daily accounting.
Job	A job is a grouping of processes that the system treats as a single entity and is identified by a unique job identifier (job ID). There are multiple accounting types, and of them, CSA is the only accounting type to organize accounting data by jobs and boot times and then place the data into a `sorted pacct` file. For non-workload management jobs, a job consists of all accounting data for a given job ID during a single boot period. A workload management job consists of the accounting data for all job IDs associated with the job's workload management request ID. Workload management jobs may span multiple boot periods. If a job is restarted, it has the same job ID associated with it during all boot periods in which it runs. Rerun workload management jobs have multiple job IDs. CSA treats all phases of a workload management job as being in the same job. **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.
Periodic accounting	Periodic (monthly) accounting further processes, reports, and summarizes the daily accounting reports to give a higher level view of how the system is being used.

	CSA lets system administrators specify the time periods for which monthly or cumulative accounting is to be run. Thus, periodic accounting can be run more than once a month, but sometimes is still referred to as monthly accounting.
Daemon accounting	Daemon accounting is the processing, organizing, and reporting of the raw accounting data, performed at the completion of daemon specific events.
Recycled data	Recycled data is data left in the raw accounting data file, saved for the next accounting report run. By default, accounting data for active jobs is recycled until the job terminates. CSA reports only data for terminated jobs unless `csarun` is invoked with the `-A` option. `csarun` places recycled data into the `/var/csa/day/pacct0` data file.

The following abbreviations and definitions are used throughout this chapter:

Abbreviation	**Definition**
MMDD	Month, day
hhmm	Hour, minute

Enabling or Disabling CSA

The following steps are required to set up CSA job accounting:

Note: Before you configure CSA on your machine, make sure that Linux jobs are installed and configured on your system. When you run the `jstat -a` command, you should see output similar to the following:

```
$ jstat -a
JID                OWNER        COMMAND
------------------ ------------ --------------------------------
0xa28052020000483d user         login -- user
0xa28052020000432f jh           /usr/sbin/sshd
```

If jobs are not installed and configured, see "Installing and Configuring Linux Kernel Jobs for Use with CSA" on page 4. For more information on the `jstat` command, see "Linux Kernel Job Overview" on page 1 and the `jstat`(1) man page.

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

   ```
   chkconfig --add csa
   ```

2. Modify the CSA configuration variables in `/etc/csa.conf` as desired. Comments in the file describe these configuration options.

3. Turn on CSA, by entering the following:

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

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

   For information on adding entries to the `crontabs` file so that the `cron`(1M) command automatically runs daily accounting, see "Setting Up CSA" on page 21.

The following steps are required to disable CSA job accounting:

1. To turn off CSA, enter the following:

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

2. To stop CSA from initiating after a system reboot, enter the `chkconfig` command as follows:

   ```
   chkconfig --del csa
   ```

CSA Files and Directories

The following sections describe the CSA files and directories.

Files in the `/var/csa` Directory

The `/var/csa` directory contains CSA data and report files within various subdirectories. `/var/csa` contains data collection files used by CSA. CSA accesses `pacct` files to process system accounting data. The following diagram shows the directory and file layout for CSA:

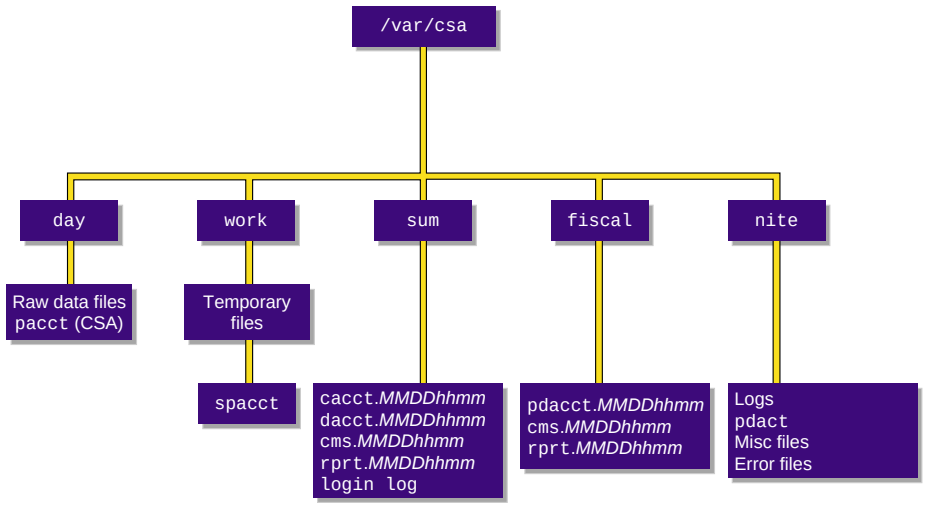

Figure 2-1 The `/var/csa` Directory

Each data and report file for CSA has a month-day-hour-minute suffix.

Note: On an extremely busy system, the data contained under `/var/csa` can potentially reach the size of multiple Megabytes. If there are many CSA transactions, you may want to consider having `/var/csa` on a disk separate from root.

Files in the `/var/csa/` Directory

The `/var/csa` directory contains the following directories:

Directory	Description
`day`	Contains the current raw accounting data files in `pacct` format.
`work`	Used by CSA as a temporary work area. Contains raw files that were moved from `/var/csa/day` at the start of a CSA daily accounting run and the `spacct` file.
`sum`	Contains the cumulative daily accounting summary files and reports created by `csarun`(8). The ASCII format is in `/var/csa/sum/rprt.`*MMDDhhmm*.

	The binary data is in `/var/csa/sum/cacct.`*MMDDhhmm*, `/var/csa/sum/cms.`*MMDDhhmm*, and `/var/csa/sum/dacct.`*MMDDhhmm*.
`fiscal`	Contains periodic accounting summary files and reports created by `csaperiod`(8). The ASCII format is in `/var/csa/fiscal/csa/rprt.`*MMDDhhmm*.
	The binary data is in `/usr/csa/fiscal/cms.`*MMDDhhmm* and `/usr/csa/fiscal/pdacct.`*MMDDhhmm*.
`nite`	Contains log files, `csarun` state, and execution times files.

Files in the `/var/csa/day` Directory

The following files are located in the `/var/csa/day` directory:

File	Description
`dodiskerr`	Disk accounting error file.
`pacct`	Process and daemon accounting data.
`pacct0`	Recycled process and daemon accounting data.
`dtmp`	Disk accounting data (ASCII) created by `dodisk`.

Files in the `/var/csa/work` Directory

The following files are located in the `/var/csa/work/`*MMDD/hhmm* directory:

File	Description
`BAD.Wpacct*`	Unprocessed accounting data containing invalid records (verified by `csaverify`(8)).
	Note: The `/var/csa/work/Wpacct*` files are generated during the execution of the `csarun`(8) command.
`Ever.tmp1`	Data verification work file.
`Ever.tmp2`	Data verification work file.
`Rpacct0`	Process and daemon accounting data to be recycled in the next accounting run.

`Wdiskcacct`	Disk accounting data (`cacct.h` format) created by `dodisk`(8) (see the `dodisk`(8) man page).
`Wdtmp`	Disk accounting data (ASCII) created by `dodisk`(8).
`Wpacct*`	Raw process and daemon accounting data.
	Note: The `/var/csa/work/Wpacct*` files are generated during the execution of the `csarun`(8) command.
`spacct`	`sorted pacct` file

Files in the `/var/csa/sum` Directory

The following data files are located in the `/var/csa/sum` directory:

File	Description
`cacct.`*MMDDhhmm*	Consolidated daily data in `cacct.h` format. This file is deleted by `csaperiod` if the `-r` option is specified.
`cms.`*MMDDhhmm*	Daily command usage data in command summary (`cms`) record format. This file is deleted by `csaperiod` if the `-r` option is specified.
`dacct.`*MMDDhhmm*	Daily disk usage data in `cacct.h` format. This file is deleted by `csaperiod` if the `-r` option is specified.
`loginlog`	Login record file created by `lastlogin`.
`rprt.`*MMDDhhmm*	Daily accounting report.

Files in the `/var/csa/fiscal` Directory

The following files are located in the `/var/csa/fiscal` directory:

File	Description
`cms.`*MMDDhhmm*	Periodic command usage data in command summary (`cms`) record format.
`pdacct.`*MMDDhhmm*	Consolidated periodic data.

`rprt.`*MMDDhhmm*	Periodic accounting report.

Files in the `/var/csa/nite` Directory

The following files are located in the `/var/csa/nite` directory:

File	Description
`active`	Used by the `csarun`(8) command to record progress and print warning and error messages. `active`*MMDDhhmm* is the same as `active` after `csarun` detects an error.
`clastdate`	Last two times `csarun` was executed; in *MMDDhhmm* format.
`dk2log`	Diagnostic output created during execution of `dodisk` (see the `cron` entry for `dodisk` in "Setting Up CSA" on page 21).
`diskcacct`	Disk accounting records in `cacct.h` format, created by `dodisk`.
`Eaddc`*MMDDhhmm*	Error/warning messages from the `csaaddc`(8) command for an accounting run done on *MMDD* at *hhmm*.
`Earc1`*MMDDhhmm*	Error/warning messages from the `csa.archive1`(8) command for an accounting run done on *MMDD* at *hhmm*.
`Earc2`*MMDDhhmm*	Error/warning messages from the `csa.archive2`(8) command for an accounting run done on *MMDD* at *hhmm*.
`Ebld.`*MMDDhhmm*	Error/warning messages from the `csabuild`(8) command for an accounting run done on *MMDD* at *hhmm*.
`Ecmd.`*MMDDhhmm*	Error/warning messages from the `csacms`(8) command when generating an ASCII report for an accounting run done on *MMDD* at *hhmm*.
`Ecms.`*MMDDhhmm*	Error/warning messages from the `csacms`(8) command when generating binary data for an accounting run done on *MMDD* at *hhmm*.

`Econ.`*MMDDhhmm*	Error/warning messages from the `csacon`(8) command for an accounting run done on *MMDD* at *hhmm*.
`Ecrep.`*MMDDhhmm*	Error/warning messages from the `csacrep`(8) command for an accounting run done on *MMDD* at *hhmm*.
`Ecrpt.`*MMDDhhmm*	Error/warning messages from the `csacrep`(8) command for an accounting run done on *MMDD* at *hhmm*.
`Edrpt.`*MMDDhhmm*	Error/warning messages from the `csadrep`(8) command for an accounting run done on *MMDD* at *hhmm*.
`Erec.`*MMDDhhmm*	Error/warning messages from the `csarecy`(8) command for an accounting run done on *MMDD* at *hhmm*.
`Euser.`*MMDDhhmm*	Error/warning messages from the `csa.user`(8) user exit for an accounting run done on *MMDD* at *hhmm*.
`Epuser.`*MMDDhhmm*	Error/warning messages from the `csa.puser`(8) user exit for an accounting run done on *MMDD* at *hhmm*.
`Ever.tmp1`*MMDDhhmm*	Output file from invalid record offsets from the `csaverify`(8) command for an accounting run done on *MMDD* at *hhmm*.
`Ever.tmp2`*MMDDhhmm*	Error/warning messages from the `csaverify`(8) command for an accounting run done on *MMDD* at *hhmm*.
`Ever.`*MMDDhhmm*	Error/warning messages from the `csaedit`(8) and `csaverify`(8) command (from the `Ever.tmp2` file) for an accounting run done on *MMDD* at *hhmm*.
`fd2log`	Diagnostic output created during execution of `csarun` (see `cron` entry for `csarun` in "Setting Up CSA" on page 21).
`lock lock1`	Used to control serial use of the `csarun`(8) comand.
`pd2log`	Diagnostic output created during execution of `csaperiod` (see `cron` entry for `csaperiod` in "Setting Up CSA" on page 21).

`pdact`	Progress and status of `csaperiod`. `pdact.`*MMDDhhmm* is the same as `pdact` after `csaperiod` detects an error.
`statefile`	Used to record current state during execution of the `csarun` command.

`/usr/sbin` and `/usr/bin` Directories

The `/usr/sbin` directory contains the following commands and shell scripts used by CSA that can be executed individually or by `cron`(1):

Command	Description
`csaaddc`	Combines *cacct* records.
`csabuild`	Organizes accounting records into job records.
`csachargefee`	Charges a fee to a user.
`csackpacct`	Checks the size of the CSA process accounting file.
`csacms`	Summarizes command usage from per-process accounting records.
`csacon`	Condenses records from the `sorted pacct` file.
`csacrep`	Reports on consolidated accounting data.
`csadrep`	Reports daemon usage.
`csaedit`	Displays and edits the accounting information.
`csagetconfig`	Searches the accounting configuration file for the specified argument.
`csajrep`	Prints a job report from the `sorted pacct` file.
`csaperiod`	Runs periodic accounting.
`csarecy`	Recycles unfinished job records into next accounting run.
`csarun`	Processes the daily accounting files and generates reports.
`csaswitch`	Checks the status of, enables or disables the different types of Comprehensive System Accounting (CSA), and switches accounting files for maintainability.

`csaverify`	Verifies that the accounting records are valid.

The `/usr/bin` directory contains the following user commands associated with CSA:

Command	Description
`csacom`	Searches and prints the CSA process accounting files.
`ja`	Starts and stops user job accounting information.

User exits allow you to tailor the `csarun` or `csaperiod` procedures to the specific needs of your site by creating scripts to perform additional site-specific processing during daily accounting. You need to create user exit files owned by `adm` or `csaacct` (`adm` for SGI ProPack 3 and `csaacct` for SGI ProPack 4) with execute permission if your site uses the accounting user exits. User exits need to be recreated when you upgrade your system. For information on setting up user exits at your site and some example user exit scripts, see "Setting up User Exits" on page 45. The `/usr/sbin` directory may contain the following scripts

Script	Description
`csa.archive1`	Site-generated user exit for `csarun`. This script saves off raw `pacct` data.
`csa.archive2`	Site-generated user exit for `csarun`. This script saves off `sorted pacct` data.
`csa.fef`	Site-generated user exit for `csarun`. This script is written by an administrator for site-specific processing.
`csa.user`	Site-generated user exit for `csarun`. This script is written by an administrator for site-specific processing.
`csa.puser`	Site-generated user exit for `csaperiod`. This script is written by an administrator for site-specific processing.

`/etc` Directory

The `/etc` directory is the location of the `csa.conf` file that contains the parameter labels and values used by CSA software.

`/etc/rc.d` Directory

The `/etc/rc.d/init.d` directory is the location of the `csaacct` file used by the `chkconfig`(8) command. Use a text editor to add any `csaswitch`(8) options to be passed to `csaswitch` during system startup only.

CSA Expanded Description

This section contains detailed information about CSA and covers the following topics:

Daily Operation Overview

When the Linux operating system is run in multiuser mode, accounting behaves in a manner similar to the following process. However, because sites may customize CSA, the following may not reflect the actual process at a particular site.

1. When CSA accounting is enabled and the system is switched to multiuser mode, the `/usr/sbin/csaswitch` (see the `csaswitch`(8) man page) command is called by `/etc/rc.d/init.d/csaacct`.
2. By default, CPU, memory, and I/O record types are enabled in `/etc/csa.conf`. However, to run workload management and tape daemon accounting, you must modify the `/etc/csa.conf` file and the appropriate subsystem. For more information, see "Setting Up CSA" on page 21.
3. The amount of disk space used by each user is determined periodically. The `/usr/sbin/dodisk` command (see `dodisk`(8)) is run periodically by the `cron` command to generate a snapshot of the amount of disk space being used by each user. The `dodisk` command should be run at most once for each time `/usr/sbin/csarun` is run (see `csarun`(8)). Multiple invocations of `dodisk` during the same accounting period write over previous `dodisk` output.
4. A fee file is created. Sites desiring to charge fees to certain users can do so by invoking `/usr/sbin/csachargefee` (see `csachargefee`(8)). Each accounting period's fee file (`/var/csa/day/fee`) is merged into the consolidated accounting records by `/usr/sbin/csaperiod` (see `csaperiod`(8)).

5. Daily accounting is run. At specified times during the day, `csarun` is executed by the `cron` command to process the current accounting data. The output from `csarun` is daily accounting files and an ASCII report.
6. Periodic (monthly) accounting is run. At a specific time during the day, or on certain days of the month, `/usr/sbin/csaperiod` (see `csaperiod`) is executed by the `cron` command to process consolidated accounting data from previous accounting periods. The output from `csaperiod` is periodic (monthly) accounting files and an ASCII report.
7. Accounting is disabled. When the system is shut down gracefully, the `csaswitch`(8) command is executed to halt all CSA process and daemon accounting.

Setting Up CSA

The following is a brief description of setting up CSA. Site-specific modifications are discussed in detail in "Tailoring CSA" on page 40. As described in this section, CSA is run by a person with superuser permissions.

1. Change the default system billing unit (SBU) weighting factors, if necessary. By default, no SBUs are calculated. If your site wants to report SBUs, you must modify the configuration file `/etc/csa.conf`.
2. Modify any necessary parameters in the `/etc/csa.conf` file, which contains configurable parameters for the accounting system.
3. If you want daemon accounting, you must enable daemon accounting at system startup time by performing the following steps:
   a. Ensure that the variables in `/etc/csa.conf` for the subsystems for which you want to enable daemon accounting are set to `on`.
   b. Set `WKMG_START` to `on` to enable workload management.
4. As root, use the `crontab`(1) command with the `-e` option to add entries similar to the following:

Note: If you do not use the `crontab`(1) command to update the `crontab` file (for example, using the `vi`(1) editor to update the file), you must signal `cron`(8) after updating the file. The `crontab` command automatically updates the `crontab` file and signals `cron`(8) when you save the file and exit the editor. For more information on the `crontab` command, see the `crontab`(1) man page.

```
0 4 *  * 1-6  if /sbin/chkconfig csaacct; then /usr/sbin/csarun 2> /var/csa/nite/fd2log; fi
0 2 *  * 4    if /sbin/chkconfig csaacct; then /usr/sbin/dodisk > /var/csa/nite/dk2log; fi
5 * *  * 1-6  if /sbin/chkconfig csaacct; then /usr/sbin/csackpacct; fi
0 5 1  * *    if /sbin/chkconfig csaacct; then /usr/sbin/csaperiod -r  \
2> /var/csa/nite/pd2log; fi
```

These entries are described in the following steps:

a. For most installations, entries similar to the following should be made in `/var/spool/cron/root` so that `cron`(8) automatically runs daily accounting:

```
0 4 *  * 1-6  if /sbin/chkconfig csaacct; then /usr/sbin/csarun 2> /var/csa/nite/fd2log; fi
0 2 *  * 4    if /sbin/chkconfig csaacct; then /usr/sbin/dodisk  > /var/csa/nite/dk2log; fi
```

The `csarun`(8) command should be executed at such a time that `dodisk` has sufficient time to complete. If `dodisk` does not complete before `csarun` executes, disk accounting information may be missing or incomplete.

For more information, see the `dodisk`(8) man page.

b. Periodically check the size of the `pacct` files. An entry similar to the following should be made in `/var/spool/cron/root`:

```
5 * *  * 1-6  if /sbin/chkconfig csaacct; then /usr/sbin/csackpacct; fi
```

The `cron` command should periodically execute the `csackpacct`(8) shell script. If the `pacct` file grows larger than 4000 1K blocks (default), `csackpacct` calls the command `/usr/sbin/csaswitch -c switch` to start a new `pacct` file. The `csackpacct` command also makes sure that there are at least 2000 1KB blocks free on the file system containing `/var/csa`. If there are not enough blocks, CSA accounting is turned off. The next time `csackpacct` is executed, it turns CSA accounting back on if there are enough free blocks.

Ensure that the ACCT_FS and MIN_BLKS variables have been set correctly in the /etc/csa.conf configuration file. ACCT_FS is the file system containing /var/csa. MIN_BLKS is the minimum number of free 1K blocks needed in the ACCT_FS file system. The default is 2000.

It is very important that csackpacct be run periodically so that an administrator is notified when the accounting file system (located in the /var/csa directory by default) runs out of disk space. After the file system is cleaned up, the next invocation of csackpacct enables process and daemon accounting. You can manually re-enable accounting by invoking csaswitch -c on.

If csackpacct is not run periodically, and the accounting file system runs out of space, an error message is written to the console stating that a write error occurred and that accounting is disabled. If you do not free disk space as soon as possible, a vast amount of accounting data can be lost unnecessarily. Additionally, lost accounting data can cause csarun to abort or report erroneous information.

c. To run monthly accounting, an entry similar to the command shown below should be made in /var/spool/cron/root. This command generates a monthly report on all consolidated data files found in /var/csa/sum/* and then deletes those data files:

```
0 5 1  * *    if /sbin/chkconfig csaacct; then /usr/sbin/csaperiod -r \
2> /var/csa/nite/pd2log; fi
```

This entry is executed at such a time that csarun has sufficient time to complete. This example results in the creation of a periodic accounting file and report on the first day of each month. These files contain information about the previous month's accounting.

5. Update the holidays file. The holidays file allows you to adust the price of system resources depending on expected demand. The file /usr/local/etc/holidays contains the prime/nonprime table for the accounting system. The table should be edited to reflect your location's holiday schedule for the year. By default, the holidays file is located in the /usr/local/etc directory. You can change this location by modifying the *HOLIDAY_FILE* variable in /etc/csa.conf. If necessary, modify the *NUM_HOLIDAYS* variable (also located in /etc/csa.conf), which sets the upper limit on the number of holidays that can be defined in *HOLIDAY_FILE*. The format of this file is composed of the following types of entries:

- Comment lines: These lines may appear anywhere in the file as long as the first character in the line is an asterisk (*).
- Version line: This line must be the first uncommented line in the file and must only appear once. It denotes that the new holidays file format is being used. This line should not be changed by the site.
- Year designation line: This line must be the second uncommented line in the file and must only appear once. The line consists of two fields. The first field is the keyword YEAR. The second field must be either the current year or the wildcard character, asterisk (*). If the year is wildcarded, the current year is automatically substituted for the year. The following are examples of two valid entries:

```
YEAR        2003
YEAR        *
```

- Prime/nonprime time designation lines: These must be uncommented lines 3, 4, and 5 in the file. The format of these lines is:

  period prime_time_start nonprime_time_start

  The variable, *period*, is one of the following: WEEKDAY, SATURDAY, or SUNDAY. The *period* can be specified in either uppercase or lowercase.

  The prime and nonprime start time can be one of two formats:

  – Both start times are 4–digit numeric values between 0000 and 2359. The *nonprime_time_start* value must be greater than the *prime_time_start* value. For example, it is incorrect to have prime time start at 07:30 A.M. and nonprime time start at 1 minute after midnight. Therefore, the following entry is wrong and can cause incorrect accounting values to be reported.

    ```
    WEEKDAY   0730    0001
    ```

    It is correct to specify prime time to start at 07:30 A.M. and nonprime time to start at 5:30 P.M. on weekdays. You would enter the following in the holiday file:

    ```
    WEEKDAY   0730    1730
    ```

  – NONE/ALL or ALL/NONE. These start times specify that the entire period is to be either all prime time or all nonprime time. To specify that the entire period is to be considered prime time, set *prime_time_start* to ALL and

nonprime_time_start to NONE. If the period is to be considered all nonprime time, set *prime_time_start* to NONE and *nonprime_time_start* to ALL. For example, to specify Monday through Friday as all prime time, you would enter the following:

```
WEEKDAY ALL NONE
```

To specify all of Sunday to be nonprime time, you would enter the following:

```
SUNDAY NONE ALL
```

- Site holidays lines: These entries follow the year designation line and have the following general format:

  day-of-year Month Day Description of Holiday

  The *day-of-year* field is either a number in the range of 1 through 366, indicating the day for a given holiday (leading white space is ignored), or it is the month and day in the *mm/dd* format. The other three fields are commentary and are not currently used by other programs. Each holiday is considered all nonprime time.

  If the holidays file does not exist or there is an error in the year designation line, the default values for all lines are used.

  If there is an error in a prime/nonprime time designation line, the entry for the erroneous line is set to a default value. All other lines in the holidays file are ignored and default values are used.

  If there is an error in a site holidays line, all holidays are ignored.

  The defaults values are as follows:

YEAR	The current year
WEEKDAY	Monday through Friday is all prime time
SATURDAY	Saturday is all nonprime time
SUNDAY	Sunday is all nonprime time
	No holidays are specified

The csarun Command

The /usr/sbin/csarun command, usually initiated by cron(1), directs the processing of the daily accounting files. csarun processes accounting records written into the pacct file. It is normally initiated by cron during nonprime hours.

The csarun command also contains four user-exit points, allowing sites to tailor the daily run of accounting to their specific needs.

The csarun command does not damage files in the event of errors. It contains a series of protection mechanisms that attempt to recognize an error, provide intelligent diagnostics, and terminate processing in such a way that csarun can be restarted with minimal intervention.

Daily Invocation

The csarun command is invoked periodically by cron. It is very important that you ensure that the previous invocation of csarun completed successfully before invoking csarun for a new accounting period. If this is not done, information about unfinished jobs will be inaccurate.

Data for a new accounting period can also be interactively processed by executing the following:

```
nohup csarun 2> /var/csa/nite/fd2log &
```

Before executing csarun in this manner, ensure that the previous invocation completed successfully. To do this, look at the files active and statefile in /var/csa/nite. Both files should specify that the last invocation completed successfully. See "Restarting csarun" on page 28.

Error and Status Messages

The csarun error and status messages are placed in the /var/csa/nite directory. The progress of a run is tracked by writing descriptive messages to the file active. Diagnostic output during the execution of csarun is written to fd2log. The lock and lock1 files prevent concurrent invocations of csarun; csarun will abort if these two files exist when it is invoked. The clastdate file contains the month, day, and time of the last two executions of csarun.

Errors and warning messages from programs called by csarun are written to files that have names beginning with E and ending with the current date and time. For

example, `Ebld.11121400` is an error file from `csabuild` for a `csarun` invocation on November 12, at 14:00.

If `csarun` detects an error, it writes a message to the `/var/log/messages` file, removes the locks, saves the diagnostic files, and terminates execution. When `csarun` detects an error, it will send mail either to `MAIL_LIST` if it is a fatal error, or to `WMAIL_LIST` if it is a warning message, as defined in the configuration file `/etc/csa.conf`.

States

Processing is broken down into separate re-entrant states so that `csarun` can be restarted. As each state completes, `/var/csa/nite/statefile` is updated to reflect the next state. When `csarun` reaches the `CLEANUP` state, it removes various data files and the locks, and then terminates.

The following describes the events that occur in each state. *MMDD* refers to the month and day `csarun` was invoked. *hhmm* refers to the hour and minute of invocation.

State	Description
`SETUP`	The current accounting file is switched via `csaswitch`. The accounting file is then moved to the `/var/csa/work/`*MMDD/hhmm* directory. File names are prefaced with `W`. `/var/csa/nite/diskcacct` is also moved to this directory.
`VERIFY`	The accounting files are checked for valid data. Records with invalid data are removed. Names of bad data files are prefixed with `BAD.` in the `/var/csa/work/`*MMDD/hhmm* directory. The corrected files do not have this prefix.
`ARCHIVE1`	First user exit of the `csarun` script. If a script named `/usr/sbin/csa.archive1` exists, it will it will be sourced by the shell using the shell . (dot) command. The . (dot) command will not execute a compiled program, but the user exit script can. You might use this user exit to archive the accounting files in `${WORK}`.
`BUILD`	The `pacct` accounting data is organized into a `sorted pacct` file.
`ARCHIVE2`	Second user exit of the `csarun` script. If a script named `/usr/sbin/csa.archive2` exists, it will be executed through the shell . (dot) command. The . (dot) command will not execute a

compiled program, but the user exit script can. You might use this exit to archive the `sorted pacct` file.

`CMS` Produces a command summary file in `cms.h` format. The `cms` file is written to `/var/csa/sum/cms.`*MMDDhhmm* for use by `csaperiod`.

`REPORT` Generates the daily accounting report and puts it into `/var/csa/sum/rprt.`*MMDDhhmm*. A consolidated data file, `/var/csa/sum/cacct.`*MMDDhhmm*, is also produced from the `sorted pacct` file. In addition, accounting data for unfinished jobs is recycled.

`DREP` Generates a daemon usage report based on the `sorted pacct` file. This report is appended to the daily accounting report, `/var/csa/sum/rprt.`*MMDDhhmm*.

`FEF` Third user exit of the `csarun` script. If a script named `/var/local/sbin/csa.fef` exists, it will be executed through the shell . (dot) command. The . (dot) command will not execute a compiled program, but the user exit script can. The `csarun` variables are available, without being exported, to the user exit script. You might use this exit to convert the `sorted pacct` file to a format suitable for a front-end system.

`USEREXIT` Fourth user exit of the `csarun` script. If a script named `/usr/sbin/csa.user` exists, it will be executed through the shell . (dot) command. The . (dot) command will not execute a compiled program, but the user exit script can. The `csarun` variables are available, without being exported, to the user exit script. You might use this exit to run local accounting programs.

`CLEANUP` Cleans up temporary files, removes the locks, and then exits.

Restarting `csarun`

If `csarun` is executed without arguments, the previous invocation is assumed to have completed successfully.

The following operands are required with `csarun` if it is being restarted:

`csarun [`*MMDD* `[`*hhmm* `[`*state*`]]]`

MMDD is month and day, *hhmm* is hour and minute, and *state* is the `csarun` entry state.

To restart csarun, follow these steps:

1. Remove all lock files, by using the following command line:

   ```
   rm -f /var/csa/nite/lock*
   ```

2. Execute the appropriate csarun restart command, using the following examples as guides:

   a. To restart csarun using the time and the state specified in clastdate and statefile, execute the following command:

      ```
      nohup csarun 0601 2> /var/csa/nite/fd2log &
      ```

      In this example, csarun will be rerun for June 1, using the time and state specified in clastdate and statefile.

   b. To restart csarun using the state specified in statefile, execute the following command:

      ```
      nohup csarun 0601 0400 2> /var/csa/nite/fd2log &
      ```

      In this example, csarun will be rerun for the June 1 invocation that started at 4:00 A.M., using the state found in statefile.

   c. To restart csarun using the specified date, time, and state, execute the following command:

      ```
      nohup csarun 0601 0400 BUILD 2> /var/csa/nite/fd2log &
      ```

      In this example, csarun will be restarted for the June 1 invocation that started at 4:00 A.M., beginning with state BUILD.

Before csarun is restarted, the appropriate directories must be restored. If the directories are not restored, further processing is impossible. These directories are as follows:

/var/csa/work/*MMDD/hhmm*
/var/csa/sum

If you are restarting at state ARCHIVE2, CMS, REPORT, DREP, or FEF , the sorted pacct file must be in /var/csa/work/*MMDD/hhmm*. If the file does not exist, csarun automatically will restart at the BUILD state. Depending on the tasks performed during the site-specific USEREXIT state, the sorted pacct file may or may not need to exist. This may or may not be acceptable.

Verifying and Editing Data Files

This section describes how to remove bad data from various accounting files.

The csaverify(8) command verifies that the accounting records are valid and identifies invalid records. The accounting file can be a pacct or sorted pacct file. When csaverify finds an invalid record, it reports the starting byte offset and length of the record. This information can be written to a file in addition to standard output. A length of -1 indicates the end of file. The resulting output file can be used as input to csaedit(8) to delete pacct or sorted pacct records.

1. The pacct file is verified with the following command line, and the following output is received:

```
$  /usr/sbin/csaverify -P pacct -o offsetfile
 /usr/sbin/csaverify: CAUTION
   readacctent(): An error was returned from the 'readpacct()' routine.
```

2. The file offsetfile from csaverify is used as input to csaedit to delete the invalid records as follows (remaining valid records are written to pacct.NEW):

   /usr/sbin/csaedit -b offsetfile -P pacct -o pacct.NEW

3. The new pacct file is reverified as follows to ensure that all the bad records have been deleted:

   /usr/sbin/csaverify -P pacct.NEW

You can use the csaedit -A option to produce an abbreviated ASCII version of pacct or sorted pacct files.

CSA Data Processing

The flow of data among the various CSA programs is explained in this section and is illustrated in Figure 2-2.

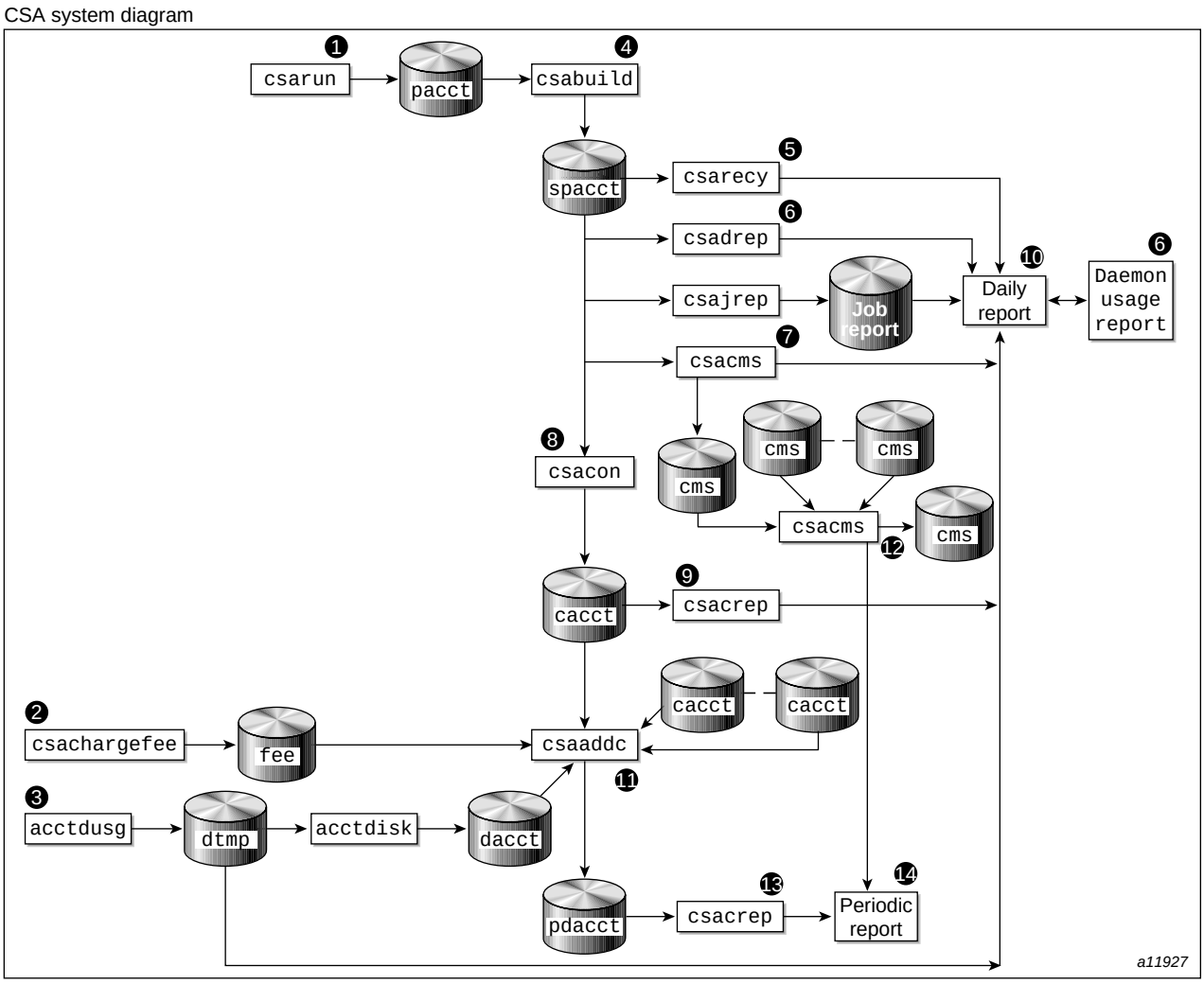

Figure 2-2 CSA Data Processing

1. Generate raw accounting files. Various daemons and system processes write to the raw `pacct` accounting files.

2. Create a fee file. Sites that want to charge fees to certain users can do so with the `csachargefee`(8) command. The `csachargefee` command creates a fee file that is processed by `csaaddc`(8).

3. Produce disk usage statistics. The `dodisk`(8) shell script allows sites to take snapshots of disk usage. `dodisk` does not report dynamic usage; it only reports the disk usage at the time the command was run. Disk usage is processed by `csaaddc`.

4. Organize accounting records into job records. The `csabuild`(8) command reads accounting records from the CSA `pacct` file and organizes them into job records by job ID and boot times. It writes these job records into the `sorted pacct` file. This `sorted pacct` file contains all of the accounting data available for each job. The configuration records in the `pacct` files are associated with the job ID 0 job record within each boot period. The information in the `sorted pacct` file is used by other commands to generate reports and for billing.

5. Recycle information about unfinished jobs. The `csarecy`(8) command retrieves job information from the `sorted pacct` file of the current accounting period and writes the records for unfinished jobs into a `pacct0` file for recycling into the next accounting period. `csabuild`(8) marks unfinished accounting jobs (those are jobs without an end-of-job record). `csarecy` takes these records from the `sorted pacct` file and puts them into the next period's accounting files directory. This process is repeated until the job finishes.

   Sometimes data for terminated jobs are continually recycled. This can occur when accounting data is lost. To prevent data from recycling forever, edit `csarun` so that `csabuild` is executed with the `-o` *nday* option, which causes all jobs older than *nday* days to terminate. Select an appropriate *nday* value (see the `csabuild` man page for more information and "Data Recycling" on page 34).

6. Generate the daemon usage report, which is appended to the daily report. `csadrep`(8) reports usage of the workload management and tape (tape is not supported in this release) daemons. Input is either from a `sorted pacct` file created by `csabuild`(8) or from a binary file created by `csadrep` with the `-o` option. The files operand specifies the binary files.

7. Summarize command usage from per-process accounting records. The `csacms`(8) command reads the `sorted pacct` files. It adds all records for processes that executed identically named commands, and it sorts and writes them to `/var/csa/sum/cms.`*MMDDhhmm*, using the `cms` format. The `csacms`(8) command can also create an ASCII file.

8. Condense records from the `sorted pacct` file. The `csacon`(8) command condenses records from the `sorted pacct` file and writes consolidated records in `cacct` format to `/var/csa/sum/cacct.`*MMDDhhmm*.
9. Generate an accounting report based on the consolidated data. The `csacrep`(8) command generates reports from data in `cacct` format, such as output from the `csacon`(8) command. The report format is determined by the value of `CSACREP` in the `/etc/csa.conf` file. Unless modified, it will report the CPU time, total `KCORE` minutes total `KVIRTUAL` minutes, block I/O wait time, and raw I/O wait time. The report will be sorted first by user ID and then by the secondary key of project ID (project ID is not supported in this release) and the headers will be printed.
10. Create the daily accounting report. The daily accounting report includes the following:
    - Consolidated information report (step 11)
    - Unfinished recycled jobs (step 5)
    - Disk usage report (step 3)
    - Daily command summary (step 7)
    - Last login information
    - Daemon usage report (step 6)
11. Combine `cacct` records. The `csaaddc`(8) command combines `cacct` records by specified consolidation options and writes out a consolidated record in `cacct` format.
12. Summarize command usage from per-process accounting records. The `csacms`(8) command reads the `cms` files created in step 7. Both an ASCII and a binary file are created.
13. Produce a consolidated accounting report. `csacrep`(8) is used to generate a report based on a periodic accounting file.
14. The periodic accounting report layout is as follows:
    - Consolidated information report
    - Command summary report

Steps 4 through 11 are performed during each accounting period by csarun(8). Periodic (monthly) accounting (steps 12 through 14) is initiated by the csaperiod(8) command. Daily and periodic accounting, as well as fee and disk usage generation (steps 2 through 3), can be scheduled by cron(8) to execute regularly. See "Setting Up CSA" on page 21, for more information.

Data Recycling

A system administrator must correctly maintain recycled data to ensure accurate accounting reports. The following sections discuss data recycling and describe how an administrator can purge unwanted recycled accounting data.

Data recycling allows CSA to properly bill jobs that are active during multiple accounting periods. By default, csarun reports data only for jobs that terminate during the current accounting period. Through data recycling, CSA preserves data for active jobs until the jobs terminate.

In the sorted pacct file, csabuild flags each job as being either active or terminated. csarecy reads the sorted pacct file and recycles data for the active jobs. csacon consolidates the data for the terminated jobs, which csaperiod uses later. csabuild, csarecy, and csacon are all invoked by csarun.

The csarun command puts recycled data in the /var/csa/day/pacct0 file.

Normally, an administrator should not have to manually purge the recycled accounting data. This purge should only be necessary if accounting data is missing. Missing data can cause jobs to recycle forever and consume valuable CPU cycles and disk space.

How Jobs Are Terminated

Interactive jobs, cron jobs, and at jobs terminate when the last process in the job exits. Normally, the last process to terminate is the login shell. The kernel writes an end-of-job (EOJ) record to the pacct file when the job terminates.

When the workload management daemon delivers a workload management request's output, the request terminates. The daemon then writes an NQ_DISP record type to the pacct accounting file, while the kernel writes an EOJ record to the pacct file.

Unlike interactive jobs, workload management requests can have multiple EOJ records associated with them. In addition to the request's EOJ record, there can be

EOJ records for net clients and checkpointed portions of the request. The net client perform workload management processing on behalf of the request.

The csabuild command flags jobs in the sorted pacct file as being terminated if they meet one of the following conditions:

- The job is an interactive, cron, or at job, and there is an EOJ record for the job in the pacct file.
- The job is a workload management request, and there is both an EOJ record for the request and an NQ_DISP record type in the pacct file.
- The job is an interactive, cron, or at job and is active at the time of a system crash. (Note that for this release jobs can not be restarted).
- The job is manually terminated by the administrator using one of the methods described in "How to Remove Recycled Data" on page 35.

Why Recycled Sessions Should Be Scrutinized

Recycling unnecessary data can consume large amounts of disk space and CPU time. The sorted pacct file and recycled data can occupy a vast amount of disk space on the file system containing /var/csa/day. Sites that archive data also require additional offline media. Wasted CPU cycles are used by csarun to reexamine and recycle the data. Therefore, to conserve disk space and CPU cycles, unnecessary recycled data should be purged from the accounting system.

Any of the following situations can cause CSA erroneously to recycle terminated jobs:

- Kernel or daemon accounting is turned off.

  The kernel or csackpacct(8) command can turn off accounting when there is not enough space on the file system containing /var/csa/day.
- Accounting files are corrupt. Accounting data can be lost or corrupted during a system or disk crash.
- Recycled data is erroneously deleted in a previous accounting period.

How to Remove Recycled Data

Before choosing to delete recycled data, you should understand the repercussions, as described in "Adverse Effects of Removing Recycled Data" on page 37. Data removal

can affect billing and can alter the contents of the consolidated data file, which is used by `csaperiod`.

You can remove recycled data from CSA in the following ways:

- Interactively execute the `csarecy -A` command. Administrators can select the active jobs that are to be recycled by running `csarecy` with the `-A` option. Users are not billed for the resources used in the jobs terminated in this manner. Deleted data is also not included in the consolidated data file.

  The following example is one way to execute `csarecy -A` (which generates two accounting reports and two consolidated files):

  1. Run `csarun` at the regularly scheduled time.
  2. Edit a copy of `/usr/sbin/csarun`. Change the `-r` option on the `csarecy` invocation line to `-A`. Also, do not redirect standard output to `${SUM_DIR}/recyrpt`. The result should be similar to the following:

```
csarecy -A -s ${SPACCT} -P ${WTIME_DIR}/Rpacct \ 2> ${NITE_DIR}/Erec.${DTIME}
```

     Since both the `-A` and `-r` options write output to `stdout`, the `-r` option is not invoked and `stdout` is not redirected to a file. As a result, the recycled job report is not generated.

  3. Execute the `jstat` command, as follows, to display a list of currently active jobs:

```
jstat -a > jstat.out
```

  4. Execute the `qstat` command to display a list of workload management requests. The `qstat` command is used for seeing whether there are requests that are not currently running. This includes requests that are checkpointed, held, queued, or waiting.

     To list all workload management requests, execute the `qstat` command, as follows, using a login that has either workload management manager or workload management operator privilege:

```
qstat -a > qstat.out
```

  5. Interactively run the modified version of `csarun`. If you execute the modified `csarun` soon after the first step is complete, little data is lost because not very much data exists.

For each active job, `csarecy` asks you if you want to preserve the job. Preserve the active and nonrunning workload management jobs found in the third and fourth steps. All other jobs are candidates for removal.

- Execute `csabuild` with the `-o` *ndays* option, which terminates all active jobs older than the specified number of days. Resource usage for these terminated jobs is reported by `csarun`, and users are billed for the jobs. The consolidated data file also includes this resource usage.

  To execute `csabuild` with the `-o` option, edit a copy of `/usr/sbin/csarun` . Add the `-o` *ndays* option to the `csabuild` invocation line. Specify for *ndays* an appropriate value for your site.

  Recycled data for currently active jobs will be removed if you specify an inappropriate value for *ndays*.

- Execute `csarun` with the `-A` option. It reports resource usage for both active and terminated jobs, so users are billed for recycled sessions. This data is also included in the consolidated data file.

  None of the data for the active jobs, including the currently active jobs, is recycled. No recycled data file is generated in the `/var/csa/day` directory.

- Remove the recycled data file from the `/var/csa/day` directory. You can delete data for all of the recycled jobs, both terminated and active, by executing the following command:

  ```
  rm /var/csa/day/pacct0
  ```

  The next time `csarun` is executed, it will not find data for any recycled jobs. Thus, users are not billed for the resources used in the recycled jobs, and this data is not included in the consolidated data file. `csarun` recycles the data for currently active jobs.

Adverse Effects of Removing Recycled Data

CSA assumes that all necessary accounting information is available to it, which means that CSA expects kernel and daemon accounting to be enabled and recycled data not to have been mistakenly removed. If some data is unavailable, CSA may provide erroneous billing information. Sites should be aware of the following facts before removing data:

- Users may or may not be billed for terminated recycled jobs. Administrators must understand which of the previously described methods cause the user to be billed

for the terminated recycled jobs. It is up to the site to decide whether or not it is valid for the user to be billed for these jobs.

For those methods that cause the user to be billed, both `csarun` and `csaperiod` report the resource usage.

- It may be impossible to reconstruct a terminated recycled job. If a recycled job is terminated by the administrator, but the job actually terminates in a later accounting period, information about the job is lost. If a user questions the resource billing, it may be extremely difficult or impossible for the administrator to correctly reassemble all accounting information for the job in question.
- Manually terminated recycled jobs may be improperly billed in a future billing period. If the accounting data for the first portion of a job has been deleted, CSA may be unable to correctly identify the remaining portion of the job. Errors may occur, such as workload management requests being flagged as interactive jobs, or workload management requests being billed at the wrong queue rate. This is explained in detail in "Workload Management Requests and Recycled Data" on page 39.
- CSA programs may detect data inconsistencies. When accounting data is missing, CSA programs may detect errors and abort.

The following table summarizes the effects of using the methods described in "How to Remove Recycled Data" on page 35.

Table 2-1 Possible Effects of Removing Recycled Data

Method	Underbilling?	Incorrect billing?	Consolidated data file
`csarecy -A`	Yes. Users are not billed for the portion of the job that was terminated by `csarecy -A`.	Possible. Manually terminated recycled jobs may be billed improperly in a future billing period.	Does not include data for jobs terminated by `csarecy -A`.
`csabuild -o`	No. Users are billed for the portion of the job that was terminated by `csabuild -o`.	Possible. Manually terminated recycled jobs may be billed improperly in a future billing period.	Includes data for jobs terminated by `csabuild -o`.

Method	Underbilling?	Incorrect billing?	Consolidated data file
`csarun -A`	No. All active and recycled jobs are billed.	Possible. All active and recycled jobs that eventually terminate may be billed improperly in a future billing period, because no data is recycled.	Includes data for all active and recycled jobs.
`rm`	Yes. All users are not billed for the portion of the job that was recycled.	Possible. All recycled jobs that eventually terminate may be billed improperly in a future billing period.	Does not include data for any recycled job.

By default, the consolidated data file contains data only for terminated jobs. Manual termination of recycled data may cause some of the recycled data to be included in the consolidated file.

Workload Management Requests and Recycled Data

For CSA to identify all workload management requests, data must be properly recycled. When an administrator manually purges recycled data for a workload management request, errors such as the following can occur:

- CSA fails to flag the job as a workload management job. This causes the request to be billed at standard rates instead of a workload management queue rate (see "Workload Management SBUs" on page 43).
- The request is billed at the wrong queue rate.
- The wrong queue wait time is associated with the request.

These errors occur because valuable workload management accounting information was purged by the administrator. Only a few workload management accounting records are written by the workload management daemon, and all of the records are needed for CSA to properly bill workload management requests.

Workload management accounting records are only written under the following circumstances:

- The workload management daemon receives a request.
- A request executes. This includes executing a request for the first time, restarting, and rerunning a request.

- A request terminates. A workload management request can terminate because it is completed, requeued, held, rerun, or migrated.
- Output is delivered.

Thus, for long running requests that span days, there can be days when no workload management data is written. Consequently, it is extremely important that accounting data be recycled. If the site administrator manually terminates recycled jobs, care must be taken to be sure that only nonexistent workload management requests are terminated.

Tailoring CSA

This section describes the following actions in CSA:

- Setting up SBUs
- Setting up daemon accounting
- Setting up user exits
- Modifying the charging of workload management jobs based on workload management termination status
- Tailoring CSA shell scripts
- Using `at`(1) instead of `cron`(8) to periodically execute `csarun`
- Allowing users without superuser permissions to run CSA
- Using an alternate configuration file

System Billing Units (SBUs)

A *system billing unit* (SBU) is a unit of measure that reflects use of machine resources. You can alter the weighting factors associated with each field in each accounting record to obtain an SBU value suitable for your site. SBUs are defined in the accounting configuration file, `/etc/csa.conf`. By default, all SBUs are set to `0.0`.

Accounting allows different periods of time to be designated either prime or nonprime time (the time periods are specified in `/usr/sbin/holidays`).

Following is an example of how the prime/nonprime algorithm works:

Assume a user uses 10 seconds of CPU time, and executes for 100 seconds of prime wall-clock time, and pauses for 100 seconds of nonprime wall-clock time. Therefore, elapsed time is 200 seconds (100+100). If

prime = *prime time* / *elapsed time*
nonprime = *nonprime time* / *elapsed time*
cputime[PRIME] = *prime* * *CPU time*
cputime[NONPRIME] = *nonprime* * *CPU time*

then

cputime[PRIME] == 5 seconds
cputime[NONPRIME] == 5 seconds

Under CSA, an SBU value is associated with each record in the `sorted pacct` file when that file is assembled by `csabuild`. Final summation of the SBU values is done by `csacon` during the creation of the `cacct` record file.

The following examples show how a site can bill different NQS or workload management queues at differing rates:

*Total SBU = (Workload management queue SBU value) * (sum of all process record SBUs + sum of all tape record SBUs)*

Process SBUs

The SBUs for process data are separated into prime and nonprime values. Prime and nonprime use is calculated by a ratio of elapsed time. If you do not want to make a distinction between prime and nonprime time, set the nonprime time SBUs and the prime time SBUs to the same value. Prime time is defined in `/usr/local/etc/holidays`. By default, Saturday and Sunday are considered nonprime time.

The following is a list of prime time process SBU weights. Descriptions and factor units for the nonprime time SBU weights are similar to those listed here. SBU weights are defined in `/etc/csa.conf`.

Value	Description
`P_BASIC`	Prime-time weight factor. `P_BASIC` is multiplied by the sum of prime time SBU values to get the final SBU factor for the process record.

P_TIME	General-time weight factor. P_TIME is multiplied by the time SBUs (made up of P_STIME, P_UTIME, P_QTIME, P_BWTIME, and P_RWTIME) to get the time contribution to the process record SBU value.
P_STIME	System CPU-time weight factor. The unit used for this weight is *billing units* per second. P_STIME is multiplied by the system CPU time.
P_UTIME	User CPU-time weight factor. The unit used for this weight is *billing units* per second. P_UTIME is multiplied by the user CPU time.
P_BWTIME	Block I/O wait time weight factor. The unit used for this weight is *billing units* per second. P_BWTIME is multiplied by the block I/O wait time.
P_RWTIME	Raw I/O wait time weight factor. The unit used for this weight is *billing units* per second. P_RWTIME is multiplied by the raw I/O wait time.
P_MEM	General-memory-integral weight factor. P_MEM is multiplied by the memory SBUs (made up of P_XMEM and P_VMEM) to get the memory contribution to the process record SBU value.
P_XMEM	CPU-time-core-physical memory-integral weight factor. The unit used for this weight is *billing units* per Mbyte-minute P_XMEM is multiplied by the core-memory integral.
P_VMEM	CPU-time-virtual-memory-integral weight factor. The unit used for this weight is *billing units* per Mbyte-minute. P_VMEM is multiplied by the virtual memory integral.
P_IO	General-I/O weight factor. P_IO is multiplied by the I/O SBUs (made up of P_BIO, P_CIO, and P_LIO) to get the I/O contribution to the process record SBU value.
P_BIO	Blocks-transferred weight factor. The unit used for this weight is *billing units* per block transferred. P_BIO is multiplied by the number of I/O blocks transferred.

`P_CIO`	Characters-transferred weight factor. The unit used for this weight is *billing units* per character transferred. `P_CIO` is multiplied by the number of I/O characters transferred.
`P_LIO`	Logical-I/O-request weight factor. The unit used for this weight is *billing units* per logical I/O request. `P_LIO` is multiplied by the number of logical I/O requests made. The number of logical I/O requests is total number of `read` and `write` system calls.

The formula for calculating the whole process record SBU is as follows:

```
PSBU = (P_TIME * (P_STIME * stime + P_UTIME * utime +
P_BWTIME * bwtime + P_RWTIME * rwtime)) + (P_MEM * (P_XMEM * coremem + P_VMEM
* virtmem)) + (P_IO * (P_BIO * bio + P_CIO * cio + P_LIO * lio));

NSBU = (NP_TIME * (NP_STIME * stime + NP_UTIME * utime
NP_BWTIME * bwtime + NP_RWTIME * rwtime)) + (NP_MEM * (NP_XMEM * coremem +
NP_VMEM * virtmem)) + (NP_IO * (NP_BIO * bio + NP_CIO * cio + NP_LIO * lio));

SBU = P_BASIC * PSBU + NP_BASIC * NSBU;
```

The variables in this formula are described as follows:

Variable	Description
stime	System CPU time in seconds
utime	User CPU time in seconds
bwtime	Block I/O wait time in seconds
rwtime	Raw I/O wait time in seconds
coremem	Core (physical) memory integral in Mbyte-minutes
virtmem	Virtual memory integral in Mbyte-minutes
bio	Number of blocks of data transferred
cio	Number of characters of data transferred
lio	Number of logical I/O requests

Workload Management SBUs

The `/etc/csa.conf` file contains the configurable parameters that pertain to workload management SBUs.

The WKMG_NUM_QUEUES parameter sets the number of queues for which you want to set SBUs (the value must be set to at least 1). Each WKMG_QUEUE *x* variable in the configuration file has a queue name and an SBU pair associated with it (the total number of queue/SBU pairs must equal WKMG_NUM_QUEUES). The queue/SBU pairs define weights for the queues. If an SBU value is less than 1.0, there is an incentive to run jobs in the associated queue; if the value is 1.0, jobs are charged as though they are non-workload management jobs; and if the SBU is 0.0, there is no charge for jobs running in the associated queue. SBUs for queues not found in the configuration file are automatically set to 1.0.

The WKMG_NUM_MACHINES parameter sets the number of originating machines for which you want to set SBUs (the value must be at least 1). Each WKMG_MACHINE *x* variable in the configuration file has an originating machine and an SBU pair associated with it (the total number of machine/SBU pairs must equal WKMG_NUM_MACHINES). SBUs for originating machines not specified in /etc/csa.conf are automatically set to 1.0.

Tape SBUs (not supported in this release)

There is a set of weighting factors for each group of tape devices. By default, there are only two groups, tape and cart. The TAPE_SBU *i* parameters in /etc/csa.conf define the weighting factors for each group. There are SBUs associated with the follpwing:

- Number of mounts
- Device reservation time (seconds)
- Number of bytes read
- Number of bytes written

Note: Tape support is not supported in this release.

Daemon Accounting

Accounting information is available from the workload management daemon. Data is written to the pacct file in the /var/csa/day directory.

In most cases, daemon accounting must be enabled by both the CSA subsystem and the daemon. "Setting Up CSA" on page 21, describes how to enable daemon

accounting at system startup time. You can also enable daemon accounting after the system has booted.

You can enable accounting for a specified daemon by using the `csaswitch` command. For example, to start tape accounting, you should do the following:

```
/usr/sbin/csaswitch -c on -n tape
```

Daemon accounting is disabled at system shutdown (see "Setting Up CSA" on page 21). It can also be disabled at any time by the `csaswitch` command when used with the `off` operand. For example, to disable workload management accounting, execute the following command:

```
/usr/sbin/csaswitch -c off -n wkmg
```

These dynamic changes using `csaswitch` are not saved across a system reboot.

Setting up User Exits

CSA accommodates the following user exits, which can be called from certain `csarun` states:

`csarun` **state**	**User exit**
`ARCHIVE1`	`/usr/sbin/csa.archive1`
`ARCHIVE2`	`/usr/sbin/csa.archive2`
`FEF`	`/var/local/sbin/csa.fef`
`USEREXIT`	`/usr/sbin/csa.user`

CSA accommodates the following user exit, which can be called from certain `csaperiod` states:

`csaperiod` **state**	**User exit**
`USEREXIT`	`/usr/sbin/csa.puser`

These exits allow an administrator to tailor the `csarun` procedure (or `csaperiod` procedure) to the individual site's needs by creating scripts to perform additional site-specific processing during daily accounting. (Note that the following comments also apply to `csaperiod`).

While executing, `csarun` checks in the `ARCHIVE1`, `ARCHIVE2`, `FEF` and `USEREXIT` states for a shell script with the appropriate name.

If the script exists, it is executed via the shell . (dot) command. If the script does not exist, the user exit is ignored. The . (dot) command will not execute a compiled program, but the user exit script can. `csarun` variables are available, without being exported, to the user exit script. `csarun` checks the return status from the user exit and if it is nonzero, the execution of `csarun` is terminated.

Some examples of user exits are as follows:

```
rain1# cd /usr/lib/acct

rain1# cat csa.archive1

#!/bin/sh
mkdir -p /tmp/acct/pacct${DTIME}
cp ${WTIME_DIR}/${PACCT}* /tmp/acct/pacct${DTIME}


rain1# cat csa.archive2

#!/bin/sh
cp ${SPACCT} /tmp/acct

rain1# cat csa.fef

#!/bin/sh
mkdir -p /tmp/acct/jobs
/usr/lib/acct/csadrep -o /tmp/acct/jobs/dbin.${DTIME} -s ${SPACCT}
/usr/lib/acct/csadrep -n -V3 /tmp/acct/jobs/dbin.${DTIME}
```

Charging for Workload Management Jobs

By default, SBUs are calculated for all workload management jobs regardless of the workload management termination code of the job. If you do not want to bill portions of a workload management request, set the appropriate *WKMG_TERM_xxxx* variable (termination code) in the `/etc/csa.conf` file to 0, which sets the SBU for this portion to 0.0. This sets the SBU for this portion to 0.0. By default, all portions of a request are billed.

The following table describes the termination codes:

Code	Description
WKMG_TERM_EXIT	Generated when the request finishes running and is no longer in a queued state.
WKMG_TERM_REQUEUE	Written for a request that is requeued.
WKMG_TERM_HOLD	Written for a request that is checkpointed and held.
WKMG_TERM_RERUN	Written when a request is rerun.
WKMG_TERM_MIGRATE	Written when a request is migrated.

Note: The above descriptions of the termination codes are very generic. Different workload managers will tailor the meaning of these codes to suit their products. LSF currently only uses the WKMG_TERM_EXIT termination code.

Tailoring CSA Shell Scripts and Commands

Modify the following variables in /etc/csa.conf if necessary:

Variable	Description
ACCT_FS	File system on which /var/csa resides. The default is /var.
MAIL_LIST	List of users to whom mail is sent if fatal errors are detected in the accounting shell scripts. The default is root and adm for SGI ProPack 3 and csaacct for SGI ProPack 4.
WMAIL_LIST	List of users to whom mail is sent if warning errors are detected by the accounting scripts at cleanup time. The default is root and adm for SGI ProPack 3 and csaacct for SGI ProPack 4.
MIN_BLKS	Minimum number of free blocks needed in ${ACCT_FS} to run csarun or csaperiod. The default is 2000 free blocks. Block size is 1024 bytes.

Using at to Execute csarun

You can use the at command instead of cron to execute csarun periodically. If your system is down when csarun is scheduled to run via cron, csarun will not be

executed until the next scheduled time. On the other hand, `at` jobs execute when the machine reboots if their scheduled execution time was during a down period.

You can execute `csarun` by using `at` in several ways. For example, a separate script can be written to execute `csarun` and then resubmit the job at a specified time. Also, an `at` invocation of `csarun` could be placed in a user exit script, `/usr/sbin/csa.user`, that is executed from the `USEREXIT` section of `csarun`. For more information, see "Setting up User Exits" on page 45.

Using an Alternate Configuration File

By default, the `/etc/csa.conf` configuration file is used when any of the CSA commands are executed. You can specify a different file by setting the shell variable `CSACONFIG` to another configuration file, and then executing the CSA commands.

For example, you would execute the following commands to use the configuration file `/tmp/myconfig` while executing `csarun`:

```
CSACONFIG=/tmp/myconfig
/usr/sbin/csarun 2> /var/csa/nite/fd2log
```

CSA Reports

You can use CSA to create accounting reports. The reports can be used to help track system usage, monitor performance, and charge users for their time on the system.

The CSA daily reports are located in the `/var/csa/sum` directory; periodic reports are located in the `/var/csa/fiscal` directory. To view the reports, go to the ASCII file `rprt.`*MMDDhhmm* in the report directories.

The CSA reports contain more detailed data than the other accounting reports. For CSA accounting, daily reports are generated by the `csarun` command. The daily report includes the following:

- disk usage statistics
- unfinished job information
- command summary data
- consolidated accounting report
- last login information

- daemon usage report

Periodic reports are generated by the `csaperiod` command. You can also create a disk usage report using the `diskusg` command.

This section describes the following reports:

CSA Daily Report

This section describes the following reports:

Consolidated Information Report

The Consolidated Information Report is sorted by user ID and then project ID (project ID is not supported in this release). The following usage values are the total amount of resources used by all processes for the specified user and project during the reporting period.

Heading	Description
PROJECT NAME	Project associated with this resource usage information (not supported in this release)
USER ID	User identifier
LOGIN NAME	Login name for the user identifier
CPU_TIME	Total accumulated CPU time in seconds
KCORE * CPU-MIN	Total accumulated amount of Kbytes of core (physical) memory used per minute of CPU time
KVIRT * CPU-MIN	Total accumulated amount of Kbytes of virtual memory used per minute of CPU time

IOWAIT BLOCK	Total accumulated block I/O wait time in seconds
IOWAIT RAW	Total accumulated raw I/O wait time in seconds

Unfinished Job Information Report

The Unfinished Job Information Report describes jobs which have not terminated and are recycled into the next accounting period.

Heading	Description
JOB ID	Job identifier
USERS	Login name of the owner of this job
PROJECT ID	Project identifier associated with this job (not supported in this release)
STARTED	Beginning time of this job

Disk Usage Report

The Disk Usage Report describes the amount of disk resource consumption by login name.

There are no column headings for this report. The first column gives the user identifier. The second column gives the login name associated with the user identifier. The third column gives the number of disk blocks used by this user.

Command Summary Report

The Command Summary Report summarizes command usage during this reporting period. The usage values are the total amount of resources used by all invocations of the specified command. Commands which were run only once are combined together in the "***other" entry. Only the first 44 command entries are displayed in the daily report. The periodic report displays all command entries.

Heading	Description
COMMAND NAME	Name of the command (program)
NUMBER OF COMMANDS	Number of times this command was executed
TOTAL KCORE-MINUTES	Total amount of Kbytes of core (physical) memory used per minute of CPU time
TOTAL KVIRT-MINUTES	Total amount of Kbytes of virtual memory used per minute of CPU time
TOTAL CPU	Total amount of CPU time used in minutes
TOTAL REAL	Total amount of real (wall clock) time used in minutes
MEAN SIZE KCORE	Average amount of core (physical) memory used in Kbytes
MEAN SIZE KVIRT	Average amount of virtual memory used in Kbytes
MEAN CPU	Average amount of CPU time used in minutes
HOG FACTOR	Total CPU time used divided by the total real time (elapsed time)
K-CHARS READ	Total number of characters read in Kbytes
K-CHARS WRITTEN	Total number of characters written in Kbytes
BLOCKS READ	Total number of blocks read
BLOCKS WRITTEN	Total number of blocks written

Last Login Report

The Last Login Report shows the last login date for each login account listed.

There are no column headings for this report. The first column is the last login date. The second column is the login account name.

Daemon Usage Report

Daemon Usage Report shows reports usage of the workload management and tape daemons (tape is not supported in this release). This report has several individual reports depending upon if there was workload management or tape daemon activity within this reporting period.

The Job Type Report gives the workload management and interactive job usage count.

Heading	Description
Job Type	Type of job (interactive or workload management)
Total Job Count	Number and percentage of jobs per job type
Tape Jobs	Number and percentage of tape jobs associated with these interactive and workload management job (not supported in this release)

The CPU Usage Report gives the workload management and interactive job usage related to CPU usage.

Heading	Description
Job Type	Type of job (interactive or workload management)
Total CPU Time	Total amount of CPU time used in seconds and percentage of CPU time
System CPU Time	Amount of system CPU time used of the total and the percentage of the total time which was system CPU time usage
User CPU Time	Amount of user CPU time used of the total and the percentage of the total time which was user CPU time usage

The workload management Queue Report gives the following information for each workload management queue.

Queue Name	Name of the workload management queue
Number of Jobs	Number of jobs initiated from this queue
CPU Time	Amount of system and user CPU times used by jobs from this queue and percentage of CPU time used
Used Tapes	How many jobs from this queue used tapes
Ave Queue Wait	Average queue wait time before initiation in seconds

Periodic Report

This section describes two periodic reports as follows:

- "Consolidated accounting report" on page 53
- "Command summary report" on page 53

Consolidated accounting report

The following usage values for the Consolidated accounting report are the total amount of resources used by all processes for the specified user and project during the reporting period.

Heading	Description
PROJECT NAME	Project associated with this resource usage information
USER ID	User identifier
LOGIN NAME	Login name for the user identifier
CPU_TIME	Total accumulated CPU time in seconds
KCORE * CPU-MIN	Total accumulated amount of Kbytes of core (physical) memory used per minute of CPU time of processes
KVIRT * CPU-MIN	Total accumulated amount of Kbytes of virtual memory used per minute of CPU time
IOWAIT BLOCK	Total accumulated block I/O wait time in seconds
IOWAIT RAW	Total accumulated raw I/O wait time in seconds
DISK BLOCKS	Total number of disk blocks used
DISK SAMPLES	Number of times disk accounting was run to obtain the disk blocks used value
FEE	Total fees charged to this user from `csachargefee`(8)
SBUs	System billing units charged to this user and project

Command summary report

The following information summarizes command usage during the defined reporting period. The usage values are the total amount of resources used by all invocations of the specified command. Unlike the daily command summary report, the periodic command summary report displays all command entries. Commands executed only

once are not combined together into an "***other" entry but are listed individually in the periodic command summary report.

Heading	Description
COMMAND NAME	Name of the command (program)
NUMBER OF COMMANDS	Number of times this command was executed
TOTAL KCORE-MINUTES	Total amount of Kbytes of core (physical) memory used per minute of CPU time
TOTAL KVIRT-MINUTES	Total amount of Kbytes of virtual memory used per minute of CPU time
TOTAL CPU	Total amount of CPU time used in minutes
TOTAL REAL	Total amount of real (wall clock) time used in minutes
MEAN SIZE KCORE	Average amount of core (physical) memory used in Kbytes
MEAN SIZE KVIRT	Average amount of virtual memory used in Kbytes
MEAN CPU	Average amount of CPU time used in minutes
HOG FACTOR	Total CPU time used divided by the total real time (elapsed time)
K-CHARS READ	Total number of characters read in Kbytes
K-CHARS WRITTEN	Total number of characters written in Kbytes
BLOCKS READ	Total number of blocks read
BLOCKS WRITTEN	Total number of blocks written

CSA Man Pages

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

This section covers these the following topics:

User-Level Man Pages

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

User-level man page	Description
csacom(1)	Searches and prints the CSA process accounting files.
ja(1)	Starts and stops user job accounting information.

Administrator Man Pages

The following administrator man page is provided with CSA software:

Administrator man page	Description
csaaddc(8)	Combines cacct records.
csabuild(8)	Organizes accounting records into job records.
csachargefee(8)	Charges a fee to a user.
csackpacct(8)	Checks the size of the CSA process accounting file.
csacms(8)	Summarizes command usage from per-process accounting records
csacon(8)	Condenses records from the sorted pacct file.
csacrep(8)	Reports on consolidated accounting data.
csadrep(8)	Reports daemon usage.
csaedit(8)	Displays and edits the accounting information.
csagetconfig(8)	Searches the accounting configuration file for the specified argument.

csajrep(8)	Prints a job report from the sorted pacct file.
csarecy(8)	Recycles unfinished jobs into the next accounting run.
csaswitch(8)	Checks the status of, enables or disables the different types of CSA, and switches accounting files for maintainability.
csaverify(8)	Verifies that the accounting records are valid.

Linux CSA Application Interface Library

The Linux CSA application interface library allows software applications to manipulate and obtain status about Linux CSA accounting methods.

Application interface man page	Description
csa_auth(3)	Checks to determine if caller has the necessary capabilities.
csa_check(3)	Checks a kernel, daemon, or record accounting state.
csa_halt(3)	Stops all accounting methods.
csa_jastart(3)	Startd job accounting.
csa_jastop(3)	Stops job accounting.
csa_kdstat(3)	Gets the kernel and daemon accounting status.
csa_rcdstat(3)	Gets the record accounting status.
csa_start(3)	Gets the user ID of a job.
csa_stop(3)	Stops specified accounting method(s).
csa_wracct(3)	Writes the accounting record to file.

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 60.

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.

Note: Steps 1 and 2 are performed automatically when the array services RPM is installed; they are included below for your information and so that you can verify that they are appropriate for your site.

---

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 `arraysvcs` on SGI ProPack 4 systems (`guest` on SGI ProPack 3 systems). 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 `arraysvcs` by default on SGI ProPack 4 systems (`guest` on SGI ProPack 3 systems), 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 76
   - "Configuring Arrays and Machines" on page 81
   - "Configuring Authentication Codes" on page 82
   - "Configuring Array Commands" on page 83

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 60, summarizes what a user needs to know and the main facilities a user has available.
- "Managing Local Processes" on page 63, reviews the conventional tools for listing and controlling processes within one node.
- "Using Array Services Commands" on page 64, describes the common concepts, options, and environment variables used by the Array Services commands.
- "Interrogating the Array" on page 68, summarizes how to use Array Services commands to learn about the Array and its workload, with examples.
- "Summary of Common Command Options" on page 66
- "Managing Distributed Processes" on page 71, 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 3-1 on page 63.

Table 3-1 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 3-2 on page 64, summarizes information about local process management.

Table 3-2 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 3-3 on page 65.

Table 3-3 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 3-4 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 3-4 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 3-5.

Table 3-5 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 arraysvcs      1.19 atop
0x1111ffff000001e9 tokyo        19816 arraysvcs      0.73 atop
0x1111ffff000001e9 disarray      1106 arraysvcs      0.47 atop
0x1111ffff000001e9 datarray      1423 arraysvcs      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 arraysvcs      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 arraysvcs@tokyo ainfo ash
Array session handle of process 13113: 0x261cffff0000145e
homegrown 181% arshell arraysvcs@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 75 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 76.)

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   arraysvcs   0:00 /bin/sh /usr/lib/array/spin
0x11110000308b2fa6  homegrown  9618  arraysvcs  0:00 sleep 5
0x11110000308b2fa6        tokyo 26021  arraysvcs   0:00 /bin/sh /usr/lib/array/spin
0x11110000308b2fa6      tokyo 26072  arraysvcs  0:00 sleep 5
0x1111ffff0000032d  homegrown  9642  arraysvcs  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  arraysvcs  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 3-6.

Table 3-6 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 66).
- 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 87.)

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 66). 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 86.

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 66).

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 73). 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 3-7. (These are described in great detail in the man page `arrayd.conf`(4).)

Table 3-7 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 3-9.

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 3-8.

Table 3-8 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 3-9.

Table 3-9 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 3-10.

Table 3-10 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 “arraysvcs.”

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 74.

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.

Note: This chapter only applies to SGI systems running SGI ProPack 3 for Linux releases. The CpuMemSets functionality in SGI ProPack 3 is contained within the new cpuset implementation on SGI ProPack 4. For more information, see Chapter 6, "Cpusets on SGI ProPack 4 for Linux" on page 121.

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 96. For more information on the Python interface, see "Managing CpuMemSets" on page 97.

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

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 4-1, 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 4-1 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 99).

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 a specified major application. This can be done using "hard" system 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 CPUs and memory of the system. 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.

For information on system partitioning, see "System Partitioning" in the *Linux Configuration and Operations 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`.

Cpusets on SGI ProPack 3 for Linux

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.

Note: This chapter only applies to SGI systems running SGI ProPack 3 for Linux releases. For information on cpusets on SGI ProPack 4 systems, see Chapter 6, "Cpusets on SGI ProPack 4 for Linux" on page 121.

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 106.

This chapter contains the following sections:

- "Cpuset System Man Pages" on page 118

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 114.
- 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 106.
- 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 114.
- 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 4, "CPU Memory Sets and Scheduling" on page 89.
- 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.

Procedure 5-1 Create the Bootcpuset

To configure the bootcpuset to be created, perform the following steps:

1. Edit or create the `/etc/bootcpuset.conf` file to include at least the following line to specify the number of nodes to be included in the bootcpuset, as follows:

   ```
   nodes=1
   ```

2. Use the `chkconfig` command to enable the bootcpuset feature, as follows:

   ```
   % chkconfig --add bootcpuset
   ```

Procedure 5-2 Destroy the Bootcpuset

To stop the bootcpuset from being created automatically during system startup, perform the following steps:

1. Remove the `/etc/bootcpuset.conf` file or edit it and comment out the `nodes=` line, as follows:

   ```
   #nodes=1
   ```

2. Use the `chkconfig` to disable the bootcpuset feature, as follows:

   ```
   % chkconfig --del bootcpuset
   ```

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

After the system boots, to make sure a boot cpuset exists, perform steps 1 though 3 in the following procedure:

Procedure 5-3 Create a New Cpuset on Your System

1. Execute the following command to determine if a bootcpuset exists on your system, as follows:

   ```
   % cpuset -Q
   ```

2. If a cpuset named *boot* exists on your system, check what CPUs are in it (CPU0 and CPU1 are used for the bootcpuset), as follows:

   ```
   % cpuset -q boot -Q
   ```

3. Check the full set of parameters for the *boot* cpuset, as follows:

   ```
   % cpuset -q boot -p
   ```

4. Create a new cpuset containing CPU 2 and CPU 3 where the `cpuset_2` file contains the following line:

   ```
   CPU 2,3
   ```

5. Set the permissions on the cpuset for all processes to join, or restrict as desired, as follows:

   ```
   % chmod 777 cpuset_2
   ```

6. Create the cpuset, as follows:

   ```
   % cpuset -q cpu_1 -c -f cpuset_2
   ```

7. Determine if the cpuset is created and check which CPUs it contains, as follows:

```
% cpuset -Q
% cpuset -q cpu_2 -Q
% cpuset -q cpu_2 -p
```

8. Try to attach a new shell to cpuset 2, as follows:

```
% cpuset -q cpu_2 -A /bin/csh
```

9. To run your job within the cpuset you created, you may also run a command similar to the following:

```
%  cpuset -q cpu_2 -A myjob
```

In order for the bootcpuset feature to appear after a system reboot, you must add the following line to the `elilo.conf` file:

```
append="cpumemset_minimal=1"
```

An example of the `elilo.conf` file is, as follows:

```
% cat /boot/efi/efi/sgi/elilo.conf
################################################
# This file is generated by System Configurator. #
################################################

# The number of deciseconds (0.1 seconds) to wait before booting
prompt
timeout=40
relocatable

# the default label to boot
default=2.4.21-sgi240c1
append="cpumemset_minimal=1"

#----- Options for KERNEL0 -----#
image=vmlinuz-2.4.21-sgi240c1
        label=2.4.21-sgi240c1
        read-only
        append="console=ttyS0,38400n8"
        root=/dev/xscsi/pci01.03.0-1/target2/lun0/part7
```

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 92 and "Tuning CpuMemSets" on page 95.

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 111.

To install the cpuset system software, see "Installing the Cpuset System" on page 117.

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 114. For restrictions that apply to CPUs belonging to cpusets, see "Restrictions on CPUs within Cpusets" on page 111.
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 111, and "Cpuset System Examples" on page 111.

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 5-1 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 114.

   For an explanation of the `MEMORY_LOCAL` flag, see "Cpuset Configuration File" on page 114.

   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 114.

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 106.

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 SGI ProPack 3" on page 145.

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

Cpusets on SGI ProPack 4 for Linux

Note: This chapter only applies to systems running SGI ProPack 4 for Linux. For information on cpusets running on SGI ProPack 3 for Linux systems, see Chapter 5, "Cpusets on SGI ProPack 3 for Linux" on page 103.

This chapter describes the cpuset facility on systems running SGI ProPack 4 for Linux and covers the following topics:

Cpuset Facility Overview

The cpuset facility is primarily a workload manager tool permitting a system administrator to restrict the number of processors and memory resources that a process or set of processes may use. A *cpuset* defines a list of CPUs and memory nodes. A process contained in a cpuset may only execute on the CPUs in that cpuset and may only allocate memory on the memory nodes in that cpuset. Essentially, cpusets provide you with a CPU and memory containers or “soft partitions” within

which you can run sets of related tasks. Using cpusets on an SGI Altix system improves cache locality and memory access times and can substantially improve an applications performance and runtime repeatability. Restraining all other jobs from using any of the CPUs or memory resources assigned to a critical job minimizes interference from other jobs on the system. For example, Message Passing Interface (MPI) jobs frequently consist of a number of threads that communicate using message passing interfaces. All threads need to be executing at the same time. If a single thread loses a CPU, all threads stop making forward progress and spin at a barrier. Cpusets can eliminate the need for a gang scheduler.

Cpusets are represented in a hierarchical virtual file system. Cpusets can be nested and they have file-like permissions.

The `sched_setaffinity`, `mbind`, and `set_mempolicy` system calls allows you to specify the CPU and memory placement for individual tasks. On smaller or limited use systems, these calls may be sufficient. For more information on these calls, see "Using Scheduling and Memory Management System Calls with Cpusets" on page 135.

The kernel cpuset facility provides additional support for system-wide management of CPU and memory resources by related sets of tasks. It provides a hierarchical structure to the resources, with filesystem-like namespace and permissions, and support for guaranteed exclusive use of resources.

The Linux 2.6 kernel provides the following support for cpusets:

- Each task has a link to a cpuset structure that specifies the CPUs and memory nodes available for its use.
- A hook in the `sched_setaffinity` system call, used for CPU placement, and hook in the `mbind` system call, used for memory placement, ensures that any requested CPU or memory node is available in that tasks cpuset.
- All tasks sharing the same placement constraints reference the same cpuset.
- These kernel cpusets are arranged in a hierarchical virtual file system, reflecting the possible nesting of "soft partitions".
- The kernel task scheduler is constrained to only schedule a task on the CPUs in that tasks cpuset.
- The kernel memory allocator is constrained to only allocate physical memory to a task from the memory nodes in that tasks cpuset.

A cpuset constrains the jobs (set of related tasks) running in it to a subset of the systems memory and CPUs. The cpuset facility allows you and your system service software to do the following:

- Create and delete named cpusets.
- Decide which CPUs and memory nodes are available to a cpuset.
- Attach a task to a particular cpuset.
- Identify all tasks sharing the same cpuset.
- Exclude any other cpuset from overlapping a given cpuset, thereby, giving the tasks running in that cpuset exclusive use of those CPUs and memory nodes.
- Perform bulk operations on all tasks associated with a cpuset, such as varying the resources available to that cpuset or hibernating those tasks in temporary favor of some other job.
- Perform sub-partitioning of system resources using hierarchical permissions and resource management.

The kernel, at system boot time, initializes one cpuset, the root cpuset, containing the entire system's CPUs and memory nodes. Subsequent user space operations can create additional cpusets.

Mounting the cpuset virtual file system (VFS) at `/dev/cpuset` exposes the kernel mechanism to user space. This VFS allows for nested resource allocations and the associated hierarchical permission model.

You can initialize and perform other cpuset operations, using any of the these three mechanisms, as follows:

- You can create, change, or query cpusets by using shell commands on `/dev/cpuset`, such as `echo`(1), `cat`(1), `mkdir`(1), or `ls`(1) as described in "Using Cpusets at the Shell Prompt" on page 129.
- You can use the `cpuset`(1) command line utility to create or destroy cpusets or to to retrieve information about existing cpusets and to attach processes to existing cpusets as described in "Cpuset Command Line Utility" on page 131.
- You can use the `libcpuset` C programming application programming interface (API) functions to query or change them from within your application as described in Appendix B, "SGI ProPack 4 Cpuset Library Functions" on page 189.

You can find information about `libcpuset` at `/usr/share/doc/packages/cpuset/cpuset.html`.

Within a single cpuset, other facilities such as `taskset`(1), `dplace`(1), first-touch memory placement, pthreads, `sched_setaffinity` and `mbind` can be used to manage processor and memory placement to a more fine-grained level.

The user–level bitmask library supports convenient manipulation of multiword bitmasks useful for CPUs and memory nodes. This bitmask library is required by and designed to work with the cpuset library. You can find information on the bitmask library on your system at `/usr/share/doc/packages/libbitmask/libbitmask.html`.

Cpuset Programming Model

The programming model for this version of cpusets is an extension of the cpuset model provided on IRIX and and earlier versions of SGI Linux. For more information on previous cpuset implementations on SGI systems, see "Cpusets on Linux versus IRIX" on page 105.

The flat name space of earlier cpuset versions on SGI systems is extended to a hierarchical name space. This will become more important as systems become larger. The name space remains visible to all tasks on a system. Once created, a cpuset remains in existence until it is deleted or until the system is rebooted, even if no tasks are currently running in that cpuset.

The key properties of a cpuset are its pathname, the list of which CPUs and memory nodes it contains, and whether the cpuset has exclusive rights to these resources.

Every task (process) in the system is attached to (running inside) a cpuset. Tasks inherit their parents cpuset attachment when forked. This binding of task to a cpuset can subsequently be changed, either by the task itself, or externally from another task, given sufficient authority.

Tasks have their CPU and memory placement constrained to whatever their containing cpuset allows. A cpuset may have exclusive rights to its CPUs and memory, which provides certain guarantees that other cpusets will not overlap.

At system boot, a top level root cpuset is created, which includes all CPUs and memory nodes on the system. The usual mount point of the cpuset file system and therefore the usual file system path to this root cpuset, is `/dev/cpuset`.

Optionally, a "boot" cpuset may be created, at `/dev/cpuset/boot`, to include typically just a one or a few CPUs and memory nodes. A typical use for a "boot" cpuset is to contain the general purpose UNIX daemons and login sessions, while reserving the rest of the system for running specific major applications on dedicated cpusets. For more information, see "Boot Cpuset" on page 136.

Moved tasks do not have the memory they might have allocated on their old nodes moved to the new nodes. On kernels that support such memory migration, use the `[optional] cpuset_migrate` to move allocated memory as well.

Cpusets have a permission structure which determines which users have rights to query, modify, and attach to any given cpuset. Rights are based on the hierarchical model provided by the underlying Linux 2.6 kernel cpuset file system.

To create a cpuset from within C language application, your program obtains a handle to a new `struct cpuset`, sets the desired attributes via that handle, and issues a `cpuset_create`() call to create the desired cpuset and bind it to the specified name. Your program can also issue calls to list by name what cpusets exist, query their properties, move tasks between cpusets, list what tasks are currently attached to a cpuset, and delete cpusets.

The names of cpusets in this C library are always relative to the root cpuset mount point, typically `/dev/cpuset`. For more information on the `libcpuset` C language application programming interface (API) functions, see Appendix B, "SGI ProPack 4 Cpuset Library Functions" on page 189.

Cpuset Directory Files

Cpusets are named, nested sets of CPUs and memory nodes. Each cpuset is represented by a directory in the cpuset virtual file system, normally mounted at `/dev/cpuset`, as described earlier.

Each cpuset directory provides the following files, that can be either read or written to:

Cpuset Directory File	Description
`cpus`	List of CPUs that tasks in the cpuset are allowed to use.
`mems`	List of memory nodes that tasks in the cpuset are allowed to use.
`tasks`	List of process IDs (PIDs) of tasks in the cpuset.

`cpu_exclusive`	Flag (0 or 1) - If set, the cpuset has exclusive use of its CPUs (no sibling or cousin cpuset may overlap CPUs).
`mem_exclusive`	Flag (0 or 1) - If set, the cpuset has exclusive use of its memory nodes (no sibling or cousin may overlap).
`notify_on_release`	Flag (0 or 1) - If set, the `/sbin/cpuset_release_agent` binary is invoked, with the name (`/dev/cpuset` relative path) of that cpuset in `argv[1]`, when the last user of it (task or child cpuset) goes away. This supports automatic cleanup of abandoned cpusets.

Cpuset Permissions

The permissions of a cpuset are determined by the permissions of the special files and directories in the cpuset file system, normally mounted at `/dev/cpuset`.

For example, a task can put itself in some other cpuset (than its current one) if it can write the `tasks` file (see "Cpuset Directory Files" on page 125) for that cpuset (requires execute permission on the encompassing directories and write permission on that tasks file).

An additional constraint is applied to requests to place some other task in a cpuset. One task may not attach another task to a cpuset unless it has permission to send that task a signal.

A task may create a child cpuset if it can access and write the parent cpuset directory. It can modify the CPUs or memory nodes in a cpuset if it can access that cpusets directory (execute permissions on the encompassing directories) and write the corresponding `cpus` or `mems` file (see "Cpuset Directory Files" on page 125).

It should be noted, however, that changes to the CPUs of a cpuset do not apply to any task in that cpuset until the task is reattached to that cpuset. If a task can write the `cpus` file, it should also be able to write the `tasks` file and might be expected to have permission to reattach the tasks therein (equivalent to permission to send them a signal).

There is one minor difference between the manner in which these permissions are evaluated and the manner in which normal file system operation permissions are evaluated. The kernel evaluates relative pathnames starting at a tasks current working directory. Even if program is operating on a cpuset file, relative pathnames

are evaluated relative to the current working directory, not relative to a tasks current cpuset. The only ways that cpuset paths relative to a tasks current cpuset can be used is if either the tasks current working directory is its cpuset (it first did a `cd`(1) `chdir`(2) to its cpuset directory beneath `/dev/cpuset`, which is a bit unusual) or if some user code converts the relative cpuset path to a full file system path (which the `cpuset`(1) command and `libcpuset`(3) library **always** do, to avoid assumptions on the current working directory). The end result is that, when using the `cpuset`(1) command or the `libcpuset` (3) library, the requesting task will require search (execute) permission on the full path to a cpuset, regardless of whether it specifies a full or relative cpuset path.

CPU Scheduling and Memory Allocation for Cpusets

This section describes CPU scheduling and memory allocation for cpusets and covers these topics:

Linux Kernel CPU and Memory Placement Settings

The Linux kernel exposes to user space three important attributes of each task that the kernel uses to control that tasks processor and memory placement, as follows:

- The cpuset path of each task, relative to the root of the cpuset file system, is available in the file `/proc/`*pid*`/cpuset`. For each task (PID), the file lists its cpuset path relative to the root of the cpuset file system.
- The actual CPU bitmask used by the kernel scheduler to determine on which CPUs a task may be scheduled is displayed in the **`Cpus_allowed`** field of the file `/proc/`*pid*`/status` for that task *pid*.
- The actual memory node bitmask used by the kernel memory allocator to determine on which memory nodes a task may obtain memory is displayed in the **`Mems_allowed`** field of the file of the file `/proc/`*pid*`/status` for that task *pid*.

Each of the above files is read-only. You can ask the kernel to make changes to these settings by using the various cpuset interfaces and the `sched_setaffinity`(2), `mbind`(2), and `set_mempolicy`(2) system calls.

The `cpus_allowed` and `mems_allowed` status file values for a task may differ from the `cpus` and `mems` values defined in the cpuset directory for the task for the following reasons:

- A task might call the `sched_setaffinity`, `mbind`, or `set_mempolicy` system calls to restrain its placement to less than its cpuset.
- Various temporary changes to `cpus_allowed` status file values are done by kernel internal code
- Attaching a task to a cpuset does not change its `mems_allowed` status file value until the next time that task needs kernel memory.
- Changing the CPUs in a cpuset does not change the `cpus_allowed` status file value of the tasks attached to the cpuset until those tasks are reattached to it (to avoid a hook in the hotpath scheduler code in the kernel).

  Use the `cpuset_reattach` routine to perform this update after a changing the CPUs allowed to a cpuset.
- If hotplug is used to remove all the CPUs or all the memory nodes in a cpuset, the tasks attached to that cpuset will have their `cpus_allowed` status file values or `mems_allowed` status file values altered to the CPUs or memory nodes when the closest ancestor to that cpuset is not empty.

Manipulating Cpusets

New cpusets are created using the `mkdir`(1) command (at the shell (see Procedure 6-1 on page 129) or in C programs (see Appendix B, "SGI ProPack 4 Cpuset Library Functions" on page 189)). Old cpusets are removed using the `rmdir`(1) commands. The `Cpus_allowed` and `Mems_allowed` status file files are accessed using `read`(2) and `write`(2) system calls or shell commands such as `cat` and `echo`.

The CPUs and memory nodes in a given cpuset are always a subset of its parent. The root cpuset has all possible CPUs and memory nodes in the system. A cpuset may be exclusive (CPU or memory) only if its parent is similarly exclusive.

Using Cpusets at the Shell Prompt

This section describes the use of cpuset using shell commands. For information on the `cpuset`(1) command line utility, see "Cpuset Command Line Utility" on page 131. For information on using the cpuset library functions, see .

When modifying the CPUs in a cpuset from the you must write the process ID (PID) of each task attached to that cpuset back into the cpusets `tasks` file. This step is handled automatically when using the `libcpuset` API. The reasons for performing this step are described in "Modifying the CPUs in a Cpuset and Kernel Processing" on page 140.

Procedure 6-1 Starting a New Job within a Cpuset

In this procedure, you will create a new cpuset called `green`, assign CPUs 2 and 3 and memory node 1 to the new cpuset, and start a subshell running in the cpuset.

To start a new job and contain it within a cpuset, perform the following steps:

1. The cpuset system is created and initialized by the kernel at system boot. You allow user space access to the cpuset system by mounting the cpuset virtual file system (VFS) at `/dev/cpuset`, as follows:

   ```
   % mkdir /dev/cpuset

   % mount -t cpuset cpuset /dev/cpuset
   ```

   Note: If the `mkdir`(1) and/or the `mount`(8) command fail, it is because they have already been performed.

2. Create the new cpuset called `green` within the `/dev/cpuset` virtual file system using the `mkdir` command, as follows:

   ```
   % cd /dev/cpuset

   % mkdir green

   % cd green
   ```

3. Use the echo command to assign CPUs 2 and 3 and memory node 1 to the `green` cpuset, as follows:

   ```
   % /bin/echo 2-3 > cpus

   % /bin/echo 1 > mems
   ```

4. Start a task that will be the "parent process" of the new job and attach the task to the new cpuset by writing its PID to the `/dev/cpuset/tasks` file for that cpuset.

   ```
   /bin/echo $$ > tasks
         sh
   ```

5. The subshell `sh` is now running in the green cpuset.

   The file `/proc/self/cpuset` shows your current cpuset, as follows:

   ```
   % cat /proc/self/cpuset
   /green
   ```

6. From this shell, you can `fork`, `exec` or `clone`(2) the job tasks. By default, any child task of this shell will also be in cpuset green. You can list the PIDs of the tasks currently in cpuset green by performing the following:

   ```
   % cat /dev/cpuset/green/tasks
   4965
   5043
   ```

   In this example, PID `4965` is your shell, and PID `5043` is the `cat` command itself.

Procedure 6-2 Removing a Cpuset from the `/dev/cpuset` Directory

To remove the cpuset `green` from the `/dev/cpuset` directory, perform the following:

1. Use the `rmdir` command to remove a directory from the `/dev/cpuset` directory, as follows:

   ```
   systemA:/dev/cpuset # rmdir green
   ```

2. To determine if you can remove the cpuset, you can perform the `cat` command on the cpuset directory `tasks` files to ensure no PIDs are listed or within an application using `libcpuset` 'C' API. You can also perform an `ls` command on the cpuset directory to ensure it has no subdirectories.

   The `green` cpuset must be emtpy in order for you to remove it, if not a message similar to the following appears:

   ```
   systemA:/dev/cpuset # rmdir green
   rmdir: 'green': Device or resource busy
   ```

Cpuset Command Line Utility

The `cpuset`(1) command is used to create and destroy cpusets, to retrieve information about existing cpusets, and to attach processes to cpusets. The `cpuset`(1) command line utility is not essential to the use of cpusets. This utility provides an alternative that may be convenient for some uses. Users of earlier versions of cpusets may find this utility familiar, though the details of the options have changed in order reflect the current implementation of cpusets.

A cpuset is defined by a cpuset configuration file and a name. For a definition of the cpuset configuration file format, see "Cpuset Text Format" on page 139. The cpuset configuration file is used to list the CPUs and memory nodes that are members of the cpuset. It also contains any additional parameters required to define the cpuset. For more information on the cpuset configuration file, see "`bootcpuset.conf` File" on page 138.

This command automatically handles reattaching tasks to their cpuset whenever necessary, as described in the `cpuset_reattach` routine in Appendix B, "SGI ProPack 4 Cpuset Library Functions" on page 189.

The cpuset command accepts the following options:

Action Options (choose exactly one):

`-c` *csname*, `--create=`*csname*	Creates cpuset named *csname* using the cpuset text format (see "Cpuset Text Format" on page 139) representation read from the commands input stream.
`-m` *csname*, `--modify=`*csname*	Modifies the existing cpuset *csname* to have the properties in the cpuset text format (see "Cpuset Text Format" on page 139) representation read from the commands input stream.
`-x` *csname*, `--remove=`*csname*	Removes the cpuset named *csname*. A cpuset may only be removed if there are no processes currently attached to it and the cpuset has no descendant cpusets.
`-d` *csname*, `--dump=`*csname*	Writes a cpuset text format representation (see "Cpuset Text

	Format" on page 139) of the cpuset named *csname* to the commands output stream.
`p` *csname*, `--procs=`*csname*	Lists to the commands output stream the processes (by `pid`) attached to the cpuset named *csname*. If the `-r` option is also specified, lists the `pid` of each process attached to any descendant of cpuset *csname*.
`-a csname, --attach=csname`	Attaches to the cpuset named *csname* the processes whose `pid`s are read from the commands input stream, one `pid` per line.
`-i` *csname*, `--invoke=`*csname*	Invokes a command in the cpuset named *csname*. If `-I` option is set, use that command and arguments, otherwise if the environment variable `$SHELL` is set, use that command, otherwise, use `/bin/sh`.
`-w` *pid*, `--which=`*pid*	Lists the name of the cpuset to which process `pid` is attached, to the commands output stream. If `pid` is zero (0), then the full cpuset path of the current task is displayed.
`-s` *csname*, `--show=`*csname*	Prints to the commands output stream the names of the cpusets below cpuset *csname*. If the `-r` option is also specified, this recursively includes *csname* and all its descendants, otherwise it just includes the immediate child cpusets of *csname*. The cpuset names are printed one per line.
`-R` *csname*, `--reattach=r`	Reattaches each task in cpuset *csname*. This is required after changing the `cpus` value of a

	cpuset, in order to get the tasks already attached to that cpuset to rebind to the changed CPU placement.
`-z` *csname*, `--size=`*csname*	Prints the size of (number of CPUs in) a cpuset to the commands output stream, as an ASCII decimal newline terminated string.
`-F` *flist*, `--family=`*flist*	Creates a family of non-overlapping child cpusets, given an *flist* of cpuset names and sizes (number of CPUs). Fails if the total sizes exceeds the size of the current cpuset. Enter cpuset names relative to the current cpuset, and their requested size, as alternating command line arguments. For example: `cpuset -F foo 2 bar 6 baz 4` This creates three child cpusets named `foo`, `bar`, and `baz`, having 2, 6, and 4 CPUs, respectively. This example will fail with an error message and a non-zero exit status if the current cpuset lacks at least 12 CPUs. These cpuset names are relative to the current cpuset and will not collide with the cpuset names descendent from other cpusets. Hence two commands, running in different cpusets, can both create a child cpuset named `foo` without a problem.

Modifier Options (may be used in any combination):

`-r, --recursive`	When used with `-p` or `-s` option, applies to all descendants recursively of the named cpuset *csname*.

`-I` *cmd*, `--invokecmd=`*cmd*	When used with the `-i` option, the command *cmd* is invoked, with any optional unused arguments. The following example invokes an interactive subshell in cpuset `foo`:
	`cpuset -i foo -I sh -- -i`
	The next example invokes a second cpuset command in cpuset `foo`, which then displays the full cpuset path of `foo`:
	`cpuset -i foo -I cpuset -- -w 0`
	Note: The double minus `--` is needed to end option parsing by the initial cpuset command.

`-f` *fname*, `--file=`*fname*	Uses file named *fname* for command input or output stream, instead of `stdin` or `stdout`.

Help Option (overrides all other options):

`-h, --help`	Displays command usage

Notes

The *csname* of "/" (slash) refers to the top cpuset, which encompasses all CPUs and memory nodes in the system. The *csname* of "." (dot) refers to the cpuset of the current task. If a *csname* begins with the "/" (slash) character, it is resolved relative to the top cpuset, otherwise it is resolved relative to the cpuset of the current task.

The 'command input stream' and 'command output stream' refer to the `stdin` (file descriptor 0) and `stdout` (file descriptor 1) of the command, unless the `-f` option is specified, in which case they refer to the file specified to `-f` option. Specifying the filename `-` to the `-f` option, as in `-f -`, is equivalent to not specifying the `-f` option at all.

Exactly **one** of the action options must be specified. They are, as follows:

`-c, -m, -x, -d, -p, -a, -i, -w, -s, -R`

The additional modifier options may be specified in any order. All modifier options are evaluated first, before the action option. If the help option is present, no action option is evaluated. The modifier options are, as follows:

```
-r, -I, -f
```

Using Scheduling and Memory Management System Calls with Cpusets

This section describes how to use he `sched_setaffinity`, `mbind`, and `set_mempolicy` system calls with cpusets.

You can use cpusets to divide a system into "soft partitions" and then for a particular multiple process application running within a cpuset, use the `sched_setaffinity`, `mbind`, and `set_mempolicy` system calls to fine tune the processor and memory placement of each individual task.

These mechanisms are integrated in the kernel and can be used together consistently. Changes to scheduling affinity (allowed CPUs) and memory policy (allowed memory nodes) are restricted by cpusets and changes to cpusets automatically update scheduling affinity and memory policy in the following manner, in order to ensure the following:

- A tasks scheduling affinity and memory policy are always contained in the cpuset to which it is attached
- A task always has a non-empty scheduling affinity (at least one allowed CPU) and non-empty memory policy (at least one allowed memory node).

How Cpusets Constrain Scheduling Affinity and Memory Policy Calls

The `sched_setaffinity`, `mbind`, and `set_mempolicy` system calls respect cpuset boundaries. If an application passes a bitmask to any of these three calls with CPU or memory node bits set that are outside the allowed cpuset, those bits are silently turned off, and the request made with whatever bits remain set, if any. If the resulting bitmask is empty, an error occurs, just as if the call had been made with an empty bitmask.

How Cpuset Changes Affect Scheduling Affinity and Memory Policy

If you change the cpuset to which a task is attached or binds that task to a different cpuset, the kernel updates that tasks scheduling affinity and memory policy in a consistent fashion. If the new cpuset overlaps with that tasks scheduling affinity, the tasks scheduling affinity is restricted to the CPUs common to both the new cpuset and to its prior scheduling affinity. If the new cpuset has no overlap, the tasks scheduling affinity is changed to allow execution on any CPU in the new cpuset. Similarly, for memory policy, the memory policy is restricted to the intersection of the new cpuset with the prior memory policy if not empty, else it is changed to allow allocation on any memory node in the new cpuset.

Batch Managers and Scheduling Affinity and Memory Policy Calls

If a more sophisticated batch manager is using these mechanisms to migrate a multiple process application, after making the necessary cpuset changes, it may want to invoke `sched_setaffinity` to rebind the individual tasks to the appropriate CPUs in the new cpuset (or the newly relocated cpuset). There is no corresponding way with the current implementation to rebind the memory policy in this situation, except by having each affected task directly invoke the necessary `mbind` and `set_mempolicy` system calls on their own behalf, which may not be practical. Memory policy in the case of a move to a non-overlapping cpuset will allow all memory nodes in a cpuset, favoring allocation on the node nearest to the one on which the task is executing when it needs more memory on which there is free memory. As noted, it may not be practical to then apply any different fine grained memory policy.

Simply changing a tasks cpuset, or rebinding it to a different cpuset, does not migrate any memory that it has already allocated. Other mechanisms are currently being developed (early 2005) to support memory migration on demand that could be used by a sophisticated batch manager to relocate a job. Since changing a tasks cpuset does immediately change the CPUs on which it is allowed to execute, changing the cpuset of an active task can result in that task executing on CPUs that are not close to the memory nodes on which it has its allocated memory, which is less efficient on NUMA systems than using local memory.

Boot Cpuset

You can use the `bootcpuset`(8) command to create a “boot”cpuset during the system boot that you can use to restrict the default placement of almost all UNIX

processes on your system. You can use the bootcpuset to reduce the interference of system processes with applications running on dedicated cpusets.

By default, the default cpuset for the `init` process, classic UNIX daemons, and user login shells is the root cpuset that contains the entire system. For systems dedicated to running particular applications, it is better to restrict `init`, the kernel daemons, and login shells to a particular set of CPUs and memory nodes called the `bootcpuset`.

This section covers the following topics:

Creating a Bootcpuset

This section describes how to create a bootcpuset.

Procedure 6-3 Creating a Bootcpuset

To create a bootcpuset, perform the following steps:

1. Create `/etc/bootcpuset.conf` file with values to restrict system processes to the CPUs and memory nodes appropriate for your system, similar to the following:

   ```
   cpus 0-1
   mems 0
   ```

2. In the `/boot/efi/efi/SuSE/elilo.conf` file (or a similar path to the `elilo.conf` file), add the following string using the instructions that follow to the append argument for the kernel your are booting:

   ```
   append="init=/sbin/bootcpuset"
   ```

   You should not directly edit `elilo.conf` because YaST and the install kernel tools may overwrite your changes when kernels are updated, and so on. Instead, edit the `/etc/elilo.conf` and run the `elilo` command. This will place an updated `elilo.conf` in `/boot/efi/efi/SuSE` and the system will know about the change too for future kernels or YaST runs.

3. Reboot your system.

   Subsequent system reboots will restrict most processes to the bootcpuset defined in `/etc/bootcpuset.conf`.

bootcpuset.conf File

The /etc/bootcpuset.conf file describes what CPUs and memory nodes are to be in the bootcpuset. The kernel boot command line option init is used to invoke the /sbin/bootcpuset binary ahead of the /sbin/init binary, using the elilo syntax: append="init=/sbin/bootcpuset"

When invoked with pid==1, the /sbin/bootcpuset binary does the following:

- Sets up a bootcpuset (configuration defined in the /etc/bootcpuset.conf file).
- Attaches itself to this bootcpuset.
- Attaches any unpinned kernel threads to it.
- Invokes an exec call to execute /sbin/init, /etc/init, /bin/init or /bin/sh.

A kernel thread is deemed to be unpinned (third bullet in the list above) if its Cpus_allowed value (as listed in that threads /proc/*pid*/status file for the Cpus_allowed field) allows running on all online CPUs. Kernel threads that are restricted to some proper subset of CPUs are left untouched, under the assumption that they have a good reason to be running on those restricted CPUs. Such kernel threads as migration (to handle moving threads between CPUs) and ksoftirqd (to handle per-CPU work off interrupts) must be pinned to each CPU or each memory node.

Comments in the /etc/bootcpuset.conf configuration file begin with the '#' character and extend to the end of the line. After stripping comments, the bootcpuset command examines the first white space separated token on each line.

If the first token on the line matches mems or mem (case insensitive match) then the second token on the line is written to the /dev/cpuset/boot/mems file.

If the first token on the line matches cpus or cpu (case insensitive match), then the second token is written to the /dev/cpuset/boot/cpus file.

If the first token in its entirety matches (case insensitive match) "verbose", the bootcpuset command prints a trace of its actions to the console. A typical such trace has 20 or 30 lines, detailing the steps taken by /sbin/bootcpuset and is useful in understanding its behavior and analyzing problems. The bootcpuset command ignores all other lines in the /etc/bootcpuset.conf configuration file.

Cpuset Text Format

Cpuset settings may be exported to and imported from text files using a text format representation of cpusets.

Unlike earlier versions of cpusets on SGI IRIX systems and some earlier versions of SGI ProPack for Linux systems, the permissions of files holding these text representations have no special significance to the implementation of cpusets. Rather, the permissions of the special cpuset files in the cpuset file system, normally mounted at `/dev/cpuset`, control reading and writing of and attaching to cpusets.

The text representation of cpusets is not essential to the use of cpusets. One can directly manipulate the special files in the cpuset file system. This text representation provides an alternative that may be convenient for some uses and a form for representing cpusets that users of earlier versions of cpusets will find familiar.

The exported cpuset text format has fewer directives than earlier IRIX and SGI ProPack for Linux versions. Additional directives may be added in the future.

The cpuset text format supports one directive per line. Comments begin with the '#' character and extend to the end of line.

After stripping comments, the first white space separated token on each remaining line selects from the following possible directives:

`cpus`	Specifies which CPUs are in this cpuset. The second token on the line must be a comma-separated list of CPU numbers and ranges of numbers.
`mems`	Specify which memory nodes are in this cpuset. The second token on the line must be a comma-separated list of memory node numbers and ranges of numbers.
`cpu_exclusive`	The `cpu_exclusive` flag is set.
`mem_exclusive`	The `mem_exclusive` flag is set.
`notify_on_release`	The `notify_on_release` flag is set

Additional unnecessary tokens on a line are quietly ignored. Lines containing only comments and white space are ignored.

The token `cpu` is allowed for `cpus` and `mem` for `mems`. Matching is case insensitive.

See the `libcpuset` routines `cpuset_import` and `cpuset_export` to handle converting the internal `struct cpuset` representation of cpusets to (export) and from (import) this text representation.

For information on manipulating cpuset text files at the shell prompt or in shell scripts using the `cpuset`(1) command, see "Cpuset Command Line Utility" on page 131.

Modifying the CPUs in a Cpuset and Kernel Processing

In order to minimize the impact of cpusets on critical kernel code, such as the scheduler, and due to the fact that the Linux kernel does not support one task updating the memory placement of another task directly, the impact on a task of changing its cpuset CPU or memory node placement or of changing to which cpuset a task is attached, is subtle and is described in the following paragraphs.

When a cpuset has its memory nodes modified, for each task attached to that cpuset, the next time that the kernel attempts to allocate a page of memory for a particular task, the kernel notices the change in the tasks cpuset, and updates its per-task memory placement to remain within the new cpusets memory placement. If the task was using memory policy `MPOL_BIND` and the nodes to which it was bound overlaps with its new cpuset, the task continues to use whatever subset of `MPOL_BIND` nodes that are still allowed in the new cpuset. If the task was using `MPOL_BIND` and now none of its `MPOL_BIND` nodes are allowed in the new cpuset, the task is essentially treated as if it was `MPOL_BIND` bound to the new cpuset (even though its NUMA placement, as queried by the `get_mempolicy`() routine, does not change). If a task is moved from one cpuset to another, the kernel adjusts the tasks memory placement, as above, the next time that the kernel attempts to allocate a page of memory for that task.

When a cpuset has its CPUs modified, each task using that cpuset **does not change** its behavior automatically. In order to minimize the impact on the critical kernel scheduling code, tasks continue to use their prior CPU placement until they are rebound to their cpuset by rewriting their PID to the `tasks` file of their cpuset. If a task is moved from one cpuset to another, its CPU placement is updated in the same way as if the tasks PID is rewritten to the `tasks` file of its current cpuset.

In summary, the memory placement of a task whose cpuset is changed is automatically updated by the kernel, on the next allocation of a page for that task but the processor placement is not updated until that tasks PID is rewritten to the `tasks` file of its cpuset. The delay in rebinding a tasks memory placement is necessary because the kernel does not support one task changing memory placement of another

task. The added user level step in rebinding a tasks CPU placement is necessary to avoid impacting the scheduler code in the kernel with a check for changes in a tasks processor placement.

NUMA Tools

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: Information about these commands and memory locality and application performance, in general, can be found in the *Linux Application Tuning Guide*.

Application Programming Interface for the Cpuset System on SGI ProPack 3

This appendix contains information about the application programming interface (API) for the Cpuset System running on SGI ProPack 3 for Linux systems. For information on the Cpuset API for SGI ProPack 4 for Linux, see Appendix B, "SGI ProPack 4 Cpuset Library Functions" on page 189.

Application Programming Interface for the Cpuset System

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;
}
```

SGI ProPack 4 Cpuset Library Functions

This appendix describes the SGI ProPack 4 `libcpuset` C programming application programming interface (API) functions and covers the following topics:

For information on the Cpuset API for SGI ProPack 3 for Linux systems, see Appendix A, "Application Programming Interface for the Cpuset System on SGI ProPack 3" on page 145.

Extensible Application Programming Interface

In order to provide for the convenient and robust extensibility of this cpuset API over time, the following function enables dynamically obtaining pointers for optional functions by name, at runtime:

```
void *cpuset_function(const char * function_name)
```

It returns a function pointer or NULL if function name is not recognized.

For maximum portability, you should not reference any optional cpuset function by explicit name.

However, if you presume that an optional function will always be available on the target systems of interest, you might decide to explicitly reference it by name, in order to improve the clarity and simplicity of the software in question.

Also to support robust extensibility, flags and integer option values have names dynamically resolved at runtime, not via preprocessor macros.

Some functions in Advanced Cpuset Library Functions are marked `[optional]`. (see "Advanced Cpuset Library Functions" on page 193). They are not available in all implementations of `libcpuset`. Additional `[optional] cpuset_*` functions may also be added in the future. Functions that are not marked `[optional]` are available on all implementations of `libcpuset.so` and can be called directly without using

`cpuset_function`(). However, any of them can also be called indirectly `via cpuset_function`().

To safely invoke an optional function, such as for example `cpuset_migrate`(), use the following call sequence:

```
/* fp has function signature of pointer to cpuset_migrate() */
      int (*fp)(struct cpuset *fromcp, struct cpuset *tocp, pid_t pid);
      fp = cpuset_function("cpuset_migrate");
      if (fp) {
              fp( ... );
      } else {
              puts ("cpuset migration not supported");
      }
```

If you invoke an `[optional]` function directly, your resulting program will not be able to link with any version of `libcpuset.so` that does not define that particular function.

Basic Cpuset Library Functions

The basic cpuset API provides functions usable from a C program for the processor and memory placement within a cpuset.

These functions enable an application to place various threads of its execution on specific CPUs within its current cpuset and perform related functions, such as, asking how large the current cpuset is and on which CPU within the current cpuset a thread is currently executing.

These functions do not provide the full power of the advanced cpuset API, but they are easier to use, and provide some common needs of multithreaded applications.

Unlike the rest of this document, the functions described in this section use cpuset relative numbering. In a cpuset of N CPUs, the relative cpu numbers range from zero to N - 1.

Memory placement is done automatically, preferring the node local to the requested CPU. Threads may only be placed on a single CPU. This avoids the need to allocate and free the bitmasks required to specify a set of serveral CPUs. These functions do not support creating or removing cpusets, only the placement of threads within an existing cpuset. This avoids the need to explicitly allocate and free cpuset structures.

Operations only apply to the current thread, avoiding the need to pass the process ID of the thread to be affected.

If more powerful capabilities are needed, use the Advanced Cpuset library functions (see "Advanced Cpuset Library Functions" on page 193). These basic functions do not provide any essential new capability. They are implemented using the advanced function and are fully interoperable with them.

On error, these routines return -1 and set `errno`. If invoked on an operating system kernel that does not support cpusets, `errno` is set to `ENOSYS`. If invoked on a system that supports cpusets, but when the cpuset file system is not currently mounted at `/dev/cpuset`, the `errno` is set to `ENODEV`.

The following inclusion and linkage provides access to the cpuset API from C code:

```
#include <cpuset.h>
/* link with -lcpuset */
```

The following functions are supported in the basic cpuset C API:

`cpuset_pin`	Pins the current thread to a CPU, preferring local memory
`cpuset_size`	Returns the number of CPUs that are in the current tasks cpuset
`cpuset_where`	Returns on which CPU in current tasks cpuset did the task most recently execute
`cpuset_unpin`	Removes the affect of `cpuset_pin`, lets the task have run of its entire cpuset

cpuset_pin

```
int cpuset_pin(int relcpu);
```

Pins the current task to execute only on the CPU `relcpu`, which is a relative CPU number within the current cpuset of that task. Also, automatically pins the memory allowed to be used by the current task to prefer the memory on that same node (as determined by the `cpuset_cpu2node` function), but to allow any memory in the cpuset if no free memory is readily available on the same node.

Return 0 on success, -1 on error. Errors include `relcpu` being too large (greater than `cpuset_size`() - 1). They also include running on a system that does not support

cpusets (ENOSYS) and running when the cpuset file system is not mounted at /dev/cpuset (ENODEV).

cpuset_size

```
int cpuset_size();
```

Returns the number of CPUs in the current tasks cpuset. The relative CPU numbers that are passed to the cpuset_pin function and that are returned by the cpuset_where function, must be between 0 and N - 1 inclusive, where N is the value returned by cpuset_size.

Returns -1 and sets errno on error. Errors include running on a system that does not support cpusets (ENOSYS) and running when the cpuset file system is not mounted at /dev/cpuset (ENODEV).

cpuset_where

```
int cpuset_where();
```

Returns the CPU number, relative to the current tasks cpuset, of the CPU on which the current task most recently executed. If a task is allowed to execute on more than one CPU, then there is no guarantee that the task is still executing on the CPU returned by cpuset_where, by the time that the user code obtains the return value.

Returns -1 and sets errno on error. Errors include running on a system that does not support cpusets (ENOSYS) and running when the cpuset file system is not mounted at /dev/cpuset (ENODEV).

cpuset_unpin

```
int cpuset_unpin();
```

Remove the CPU and Memory pinning affects of any previous cpuset_pin call, allowing the current task to execute on any CPU in its current cpuset and to allocate memory on any memory node in its current cpuset. Return -1 on error, 0 on success.

Returns -1 and sets errno on error. Errors include running on a system that does not support cpusets (ENOSYS) and running when the cpuset file system is not mounted at /dev/cpuset (ENODEV).

Advanced Cpuset Library Functions

The advanced cpuset API provides functions usable from a C language application for managing cpusets on a system-wide basis.

These functions primarily deal with the following three entities:

- `struct cpuset *` structure
- system cpusets
- tasks

The `struct cpuset *` structure provides a transient in-memory structure used to build up a description of an existing or desired cpuset. These structures can be allocated, freed, queried, and modified.

Actual kernel cpusets are created under the `/dev/cpuset` directory, which is the usual mount point of the kernel's virtual cpuset filesystem. These cpusets are visible to all tasks in the system, and persist until the system is rebooted or until the cpuset is explicitly deleted. These cpusets can be created, deleted, queried, modified, listed, and examined.

Every task (also known as a process) is bound to exactly one cpuset at a time. You can list which tasks are bound to a given cpuset, and to which cpuset a given task is bound. You can change to which cpuset a task is bound.

The primary attributes of a cpuset are its lists of CPUs and memory nodes. The scheduling affinity for each task, whether set by default or explicitly by the `sched_setaffinity`() system call, is constrained to those CPUs that are available in that tasks cpuset. The NUMA memory placement for each task, whether set by default or explicitly by the `mbind`() system call, is constrained to those memory nodes that are available in that tasks cpuset. This provides the essential purpose of cpusets - to constrain the CPU and Memory usage of tasks to specified subsets of the system.

The other essential attribute of a cpuset is its pathname beneath `/dev/cpuset`. All tasks bound to the same cpuset pathname can be managed as a unit, and this hierarchical name space describes the nested resource management and hierarchical permission space supported by cpusets. Also, this hierarchy is used to enforce strict exclusion, using the following rules:

- A cpuset may only be marked strictly exclusive for CPU or memory if its parent is also.

- A cpuset may not make any CPUs or memory nodes available that are not also available in its parent.
- If a cpuset is exclusive for CPU or memory, it may not overlap CPUs or memory with any of its siblings.

The combination of these rules enables checking for strict exclusion just by making various checks on the parent, siblings, and existing child cpusets of the cpuset being changed, without having to check all cpusets in the system.

On error, some of these routines return -1 or NULL and set `errno`. If one of the routines below that requires cpuset kernel support or the cpuset file system mounted is invoked on an operating system kernel that does not support cpusets, then that routine returns failure and `errno` is set to `ENOSYS`. If invoked on a system that supports cpusets, but when the cpuset file system is not currently mounted at `/dev/cpuset`, it returns failure and `errno` is set to `ENODEV`.

The following inclusion and linkage provides access to the cpuset API from C code:

```
#include <bitmask.h>
#include <cpuset.h>
/* link with -lcpuset */
```

The following functions are supported in the advanced cpuset C API:

Allocate and free `struct cpuset *` structure

- `cpuset_alloc` - Returns handle to newly allocated `struct cpuset *` structure
- `cpuset_free` - Discards no longer needed `struct cpuset *` structure

Lengths of CPUs and Memory Nodes bitmasks - needed to allocate bitmasks

- `cpuset_cpus_nbits` - Number of bits needed for a CPU bitmask on current system
- `cpuset_mems_nbits` - Number of bits needed for a Memory bitmask on current system

Set various attributes of a `struct cpuset *` Structure

- `cpuset_setcpus` - Specifies CPUs in cpuset
- `cpuset_setmems` - Specifies memory nodes in cpuset
- `cpuset_set_iopt` - Specifies an integer value option of cpuset

- `cpuset_set_sopt` - [optional] Specifies a string value option of cpuset

Query various attributes of a `struct cpuset *` Structure

- `cpuset_getcpus` - Queries CPUs in cpuset
- `cpuset_getmems` - Queries Memory Nodes in cpuset
- `cpuset_cpus_weight` - Number of CPUs in a cpuset
- `cpuset_mems_weight` - Number of memory nodes in a cpuset
- `cpuset_get_iopt` - Query an integer value option of cpuse
- `cpuset_set_sopt` - [optional] Species a string value option of cpuset

Local CPUs and Memory Nodes

- `cpuset_localcpus` - Queries the CPUs local to specified memory nodes
- `cpuset_localmems` - Queries the memory nodes local to specified CPUs
- `cpuset_cpumemdist` - [optional] Hardware distance from CPU to Memory Node
- `cpuset_cpu2node` - Returns number of Memory Node closed to specified CPU

Create, delete, query, modify, list, and examine cpusets

- `cpuset_create` - Creates a named cpuset as specified by `struct cpuset *` structure
- `cpuset_delete` - Deletes the specified cpuset (if empty)
- `cpuset_query` - Sets the `struct cpuset` structure to settings of specified cpuset
- cpuset_modify - Modifies the settings of a cpuset to those specified in a `struct cpuset` structure
- `cpuset_getcpusetpath` - Gets path of a tasks (0 for current) cpuset.
- `cpuset_cpusetofpid` - Sets the `struct cpuset` structure to settings of cpuset of specified task

List tasks (pids) currently attached to a cpuset

- `cpuset_init_pidlist` - Initializes a list of tasks (pids) attached to a cpuset
- `cpuset_pidlist_length` - Returns number of elements in a list of pid

- `cpuset_get_pidlist` - Returns i'th element of a list of pids
- `cpuset_free_pidlist` - Deallocate a list of pid

Attach tasks to cpusets

- `cpuset_move` - Moves task (0 for current) to a cpuset
- `cpuset_move_all` - Moves all tasks in a list of pids to a cpuset
- `cpuset_migrate` - [optional] Moves a task and its memory to a cpuset
- `cpuset_migrate_all` - [optional] Moves all tasks with memory in a list of pids to a cpus
- `cpuset_reattach` - Rebinds `cpus_allowed` of each task in a cpuset after changing its cpus

Map between cpuset relative and system-wide CPU and memory node numbers

- `cpuset_c_rel_to_sys_cpu` - Maps cpuset relative CPU number to system wide number
- `cpuset_c_sys_to_rel_cpu` - Maps system-wide CPU number to cpuset relative number
- `cpuset_c_rel_to_sys_mem` - Maps cpuset relative memory node number to system wide number
- `cpuset_c_sys_to_rel_mem` - Maps system-wide memory node number to cpuset relative number
- `cpuset_p_rel_to_sys_cpu` - Maps task cpuset relative CPU number to system wide number
- `cpuset_p_sys_to_rel_cpu` - Maps system-wide CPU number to task cpuset relative number
- `cpuset_p_rel_to_sys_mem` - Mapa task cpuset relative memory node number to system-wide number
- `cpuset_p_sys_to_rel_mem` - Maps system-wide memory node number to task cpuset relative number

Bind to a CPU or memory node within the current cpuset

- `cpuset_cpubind` - Binds to a single CPU within a cpuset (uses `sched_setaffinity`(2))
- `cpuset_latestcpu` - Most recent CPU on which a task has executed
- `cpuset_membind` - Binds to a single memory node within a cpuset (uses `set_mempolicy`(2))

Export cpuset settings to a regular file and import them from a regular file

- `cpuset_export` - Exports cpuset settings to a text file
- `cpuset_import` - Imports cpuset settings from a text file

Support calls to [optional] `cpuset_*` API routines

- `cpuset_function` - Return pointer to a `libcpuset.so` function, or NULL

Cpuset Library Functions Calling Sequence

A typical calling sequence would use the above functions in the following order to create a new cpuset named "xyz" and attach itself to it, as follows:

```
struct cpuset *cp = cpuset_alloc();
various cpuset_set*(cp, ...) calls
cpuset_create(cp, "xyz");
cpuset_free(cp);
cpuset_move(0, "xyz");
```

Note: Some functions are marked [optional]. For an explanation, see "Extensible Application Programming Interface" on page 189.

`cpuset_alloc`

```
struct cpuset *cpuset_alloc();
```

Creates, initializes, and returns a handle to a `struct cpuset` structure, that is an opaque data structure used to describe a cpuset.

After obtaining a `struct cpuset` handle with this call, you can use the various `cpuset_set`() methods to specify which CPUs and memory nodes are in the cpuset

and other attributes. Then, you can create such a cpuset with the `cpuset_create`() call and free cpuset handles with the `cpuset_free`() call.

The `cpuset_alloc` function returns a zero pointer (NULL) and sets `errno` in the event that `malloc`(3) fails. See the `malloc`(3) man page for possible values of `errno` (`ENOMEM` being the most likely).

cpuset_free

```
struct cpuset *cpuset_alloc();
```

Frees the memory associated with a `struct cpuset` handle, that must have been returned by a previous `cpuset_alloc`() call. If `cp` is NULL, no operation is performed.

cpuset_cpus_nbits

```
int cpuset_cpus_nbits();
```

Return the number of bits in a CPU bitmask on current system. Useful when using `bitmask_alloc`() call to allocate a CPU mask. Some other routines below return `cpuset_cpus_nbits`() as an out-of-bounds indicator.

cpuset_mems_nbits

```
int cpuset_mems_nbits();
```

Returns the number of bits in a memory node bitmask on current system. Useful when using a `bitmask_alloc`() call to allocate a memory mode mask. Some other routines below return `cpuset_mems_nbits`() as an out-of-bounds indicator.

cpuset_setcpus

```
int cpuset_setcpus(struct cpuset *cp, const struct bitmask
*cpus);
```

Given a bitmask of CPUs, the `cpuset_setcpus`() call sets the specified cpuset `cp` to include exactly those CPUs.

Returns 0 on success, else -1 on error, setting `errno`. This routine can fail if `malloc`(3) fails. See the `malloc`(3) man page for possible values of `errno` (`ENOMEM` being the most likely).

cpuset_setmems

```
void cpuset_setmems(struct cpuset *cp, const struct bitmask
*mems);
```

Given a bitmask of memory nodes, the `cpuset_setmems`() call sets the specified cpuset `cp` to include exactly those memory nodes.

Returns 0 on success, else -1 on error, setting `errno`. This routine can fail if `malloc`(3) fails. See the `malloc`(3) man page for possible values of `errno` (`ENOMEM` being the most likely).

cpuset_set_iopt

```
int cpuset_set_iopt(struct cpuset *cp, const char *optionname,
int value);
```

Sets cpuset integer valued option `optionname` to specified integer value. Returns 0 if `optionname` is recognized and value is an allowed value for that option. Returns -1 if `optionname` is recognized, but value is not allowed. Returns -2 if `optionname` is not recognized. Boolean options accept any non-zero value as equivalent to a value of one (1).

cpuset_set_sopt

```
int cpuset_set_sopt(struct cpuset *cp, const char *optionname,
const char *value);
```

Sets cpuset string valued option `optionname` to specified string value.

Returns 0 if `optionname` is recognized and value is an allowed value for that option. Returns -1 if `optionname` is recognized, but value is not allowed. Returns -2 if `optionname` is not recognized.

This is an [optional] function. Use the `cpuset_function`() to invoke it, as follows:

```
/* fp has function signature of pointer to cpuset_set_sopt() */
int (*fp)(struct cpuset *cp, const char *optionname, const char *value);
fp = cpuset_function("cpuset_set_sopt");
if (fp) {
    fp( ... );
} else {
    puts ("cpuset_set_sopt not supported");
}
```

cpuset_getcpus

```
int cpuset_getcpus(const struct cpuset *cp, struct bitmask
*cpus);
```

Queries CPUs in cpuset `cp`, by writing them to the `bitmask` `cpus`. Pass `cp == NULL` to query the current tasks cpuset.

If the memory nodes have not been set in cpuset `cp`, then no operation is performed, -1 is returned, and `errno` is set to `EINVAL`.

Returns 0 on success, else -1 on error, setting `errno`. This routine can fail if `malloc`(3) fails. See the `malloc`(3) man page for possible values of `errno` (`ENOMEM` being the most likely).

cpuset_getmems

```
int cpuset_getmems(const struct cpuset *cp, struct bitmask
*mems);
```

Queries memory nodes in cpuset `cp`, by writing them to the `bitmask` `mems`. Pass `cp == NULL` to query the current tasks cpuset.

If the memory nodes have not been set in cpuset `cp`, then no operation is performed, -1 is returned, and `errno` is set to `EINVAL`.

Returns 0 on success, else -1 on error, setting `errno`. This routine can fail if `malloc`(3) fails. See the `malloc`(3) man page for possible values of `errno` (`ENOMEM` being the most likely).

cpuset_cpus_weight

```
int cpuset_cpus_weight(const struct cpuset *cp);
```

Queries the number of CPUs in cpuset `cp`. Pass `cp == NULL` to query the current tasks cpuset.

If the CPUs have not been set in cpuset `cp`, then zero(0) is returned.

cpuset_mems_weight

```
int cpuset_mems_weight(const struct cpuset *cp);
```

Queries the number of memory nodes in cpuset `cp`. Pass `cp == NULL` to query the current tasks cpuset.

If the memory nodes have not been set in cpuset `cp`, then zero (0) is returned.

cpuset_get_iopt

```
int cpuset_get_iopt(const struct cpuset *cp, const char
*optionname);
```

Queries the value of integer option `optionname` in cpuset `cp`.

Returns value of `optionname` is recognized, else returns -1. Integer values in an uninitialized cpuset have value 0.

cpuset_get_sopt

```
const char *cpuset_get_sopt(const struct cpuset *cp, const char
*optionname);
```

Queries the value of string option `optionname` in cpuset `cp`.

Returns pointer to value of `optionname` is recognized, else returns NULL. String values in an uninitialized cpuset have value NULL.

This is an [optional] function. Use `cpuset_function`() to invoke it, as follows:

```
/* fp has function signature of pointer to cpuset_get_sopt() */
int (*fp)(struct cpuset *cp, const char *optionname);
```

```
fp = cpuset_function("cpuset_get_sopt");
if (fp) {
    fp( ... );
} else {
    puts ("cpuset_get_sopt not supported");
}
```

cpuset_localcpus

```
int cpuset_localcpus(const struct bitmask *mems, struct bitmask
*cpus);
```

Queries the CPUs local to specified memory nodes `mems`, by writing them to the `bitmask` `cpus`.

Returns 0 on success, -1 on error, setting `errno`.

cpuset_localmems

```
int cpuset_localmems(const struct bitmask *cpus, struct bitmask
*mems);
```

Queries the memory nodes local to specified CPUs `cpus`, by writing them to the `bitmask` `mems`.

Returns 0 on success, -1 on error, setting `errno`.

cpuset_cpumemdist

```
unsigned int cpuset_cpumemdist(int cpu, int mem);
```

Distance between hardware CPU `cpu` and memory node `mem`, on a scale which has the closest distance of a CPU to its local memory valued at ten (10), and other distances more or less proportional. Distance is a rough metric of the bandwidth and delay combined, where a higher distance means lower bandwidth and longer delays.

If either `cpu` or `mem` is not known to the current system, or if any internal error occurs while evaluating this distance, or if a node has no CPUs nor memory (I/O only), then the distance returned is `UCHAR_MAX` (from `limits.h`).

These distances are obtained from the systems ACPI SLIT table, and should conform to: System Locality Information Table Interface Specification Version 1.0, July 25, 2003

This is an [optional] function. Use `cpuset_function()` to invoke it.

cpuset_cpu2node

```
int cpuset_cpu2node(int cpu);
```

Returns number of memory node closest to CPU `cpu`. For NUMA architectures (as of this writing), this commonly would be the number of the node on which `cpu` is located. If an architecture did not have memory on the same node as a CPU, it would be the node number of the memory node closest to or preferred by that `cpu`.

cpuset_create

```
int cpuset_create(const char *cpusetpath, const struct *cp);
```

Creates a cpuset at the specified `cpusetpath`, as described in the provided `struct cpuset *cp` structure. The parent cpuset of that pathname must already exist. The parameter `cp` refers to a handle obtained from a `cpuset_alloc()` call. If the parameter `cpusetpath` starts with a slash (`/`) character, this a path relative to `/dev/cpuset`, otherwise, it is relative to the current tasks cpuset.

Returns 0 on success, else -1 on error, setting `errno`.

This routine can fail if `malloc`(3) fails. See the `malloc`(3) man page for possible values of `errno` (`ENOMEM` being the most likely).

cpuset_delete

```
int cpuset_delete(const char *cpusetpath);
```

Deletes a cpuset at the specified `cpusetpath`. The cpuset of that pathname must already exist, be empty (no child cpusets) and be unused (no using tasks).

If the parameter `cpusetpath` starts with a slash (`/`) character, this a path relative to `/dev/cpuset`, otherwise, it is relative to the current tasks cpuset.

Returns 0 on success, else -1 on error, setting `errno`.

cpuset_query

```
int cpuset_query(struct cpuset *cp, const char *cpusetpath);
```

Set `struct cpuset` structure to settings of cpuset at specified path `cpusetpath`. `struct cpuset *cp` must have been returned by a previous `cpuset_alloc`() call. Any previous settings of `cp` are lost.

If the parameter `cpusetpath` starts with a slash (/) character, this a path relative to `/dev/cpuset`, otherwise, it is relative to the current tasks cpuset.

Returns 0 on success, or -1 on error, setting `errno`. Errors include `cpusetpath` not referencing a valid cpuset path relative to `/dev/cpuset`, or the current task lacking permission to query that cpuset.

cpuset_modify

```
int cpuset_modify(const char *cpusetpath, const struct *cp);
```

Modify the cpuset at the specified `cpusetpath`, as described in the provided `struct cpuset *cp`. The cpuset at that pathname must already exist. The parameter `cp` refers to a handle obtained from a `cpuset_alloc`() call.

If the parameter `cpusetpath` starts with a slash (/) character, this a path relative to `/dev/cpuset`, otherwise, it is relative to the current tasks cpuset.

Returns 0 on success, else -1 on error, setting `errno`.

cpuset_getcpusetpath

```
char *cpuset_getcpusetpath(pid_t pid, char *buf, size_t size);
```

The `cpuset_getcpusetpath`() function copies an absolute pathname of the cpuset to which task of process ID `pid` is attached, to the array pointed to by `buf`, which is of length size. Use `pid == 0` for the current process.

The provided path is relative to the cpuset file system mount point.

If the cpuset path name would require a buffer longer than size elements, NULL is returned, and `errno` is set to `ERANGE` an application should check for this error, and allocate a larger buffer if necessary.

Returns NULL on failure with `errno` set accordingly, and `buf` on success. The contents of `buf` are undefined on error.

ERRORS are, as follows:

`EACCES`	Permission to read or search a component of the file name was denied.
`EFAULT`	`buf` points to a bad address.
`ESRCH`	The `pid` does not exist.
`E2BIG`	Larger buffer needed.
`ENOSYS`	Kernel does not support cpusets.

cpuset_cpusetofpid

```
int cpuset_cpusetofpid(struct cpuset *cp, int pid);
```

Set `struct cpuset` to settings of `cpuset` to which specified task `pid` is attached. `struct cpuset *cp` must have been returned by a previous `cpuset_alloc`() call. Any previous settings of `cp` are lost.

Returns 0 on success, or -1 on error, setting `errno`.

ERRORS are, as follows:

`EACCES`	Permission to read or search a component of the file name was denied.
`EFAULT`	`buf` points to a bad address.
`ESRCH`	The `pid` does not exist.
`ERANGE`	Larger buffer needed.
`ENOSYS`	Kernel does not support cpusets.

cpuset_cpusetofpid

```
int cpuset_cpusetofpid(struct cpuset *cp, int pid);
```

Sets the `struct cpuset` structure to settings of `cpuset` to which specified task `pid` is attached. `struct cpuset *cp` must have been returned by a previous `cpuset_alloc`() call. Any previous settings of `cp` are lost.

Returns 0 on success, or -1 on error, setting `errno`.

ERRORS are, as follows:

`EACCES`	Permission to read or search a component of the file name was denied.
`EFAULT`	`buf` points to a bad address.
`ESRCH`	The `pid` does not exist.
`ERANGE`	Larger buffer needed.
`ENOSYS`	Kernel does not support cpusets.

cpuset_init_pidlist

```
struct cpuset_pidlist *cpuset_init_pidlist(const char
*cpusetpath, int recursiveflag);
```

Initializes and returns a list of tasks (`pid`s) attached to cpuset `cpusetpath`. If recursiveflag is zero, include only the tasks directly in that cpuset, otherwise, include all tasks in that cpuset or any descendant thereof.

Beware that tasks can come and go from a cpuset, after this call is made.

If the parameter cpusetpath starts with a slash (/) character, then this a path relative to `/dev/cpuset`, otherwise, it is relative to the current tasks cpuset.

On error, return NULL and set `errno`.

cpuset_pidlist_length

```
int cpuset_pidlist_length(const struct cpuset_pidlist *pl);
```

Returns the number of elements in `cpuset_pidlist` `pl`.

cpuset_free_pidlist

```
void cpuset_freepidlist(struct cpuset_pidlist *pl);
```

Deallocates a list of attached `pid`s

cpuset_move

```
int cpuset_move(pid_t p, const char *cpusetpath);
```

Moves the task whose process ID is `p` to cpuset `cpusetpath`.

If `pid` is zero, then the current task is moved. If the parameter `cpusetpath` starts with a slash (/) character, this a path relative to `/dev/cpuset`, otherwise, it is relative to the current tasks cpuset.

Returns 0 on success, else -1 on error, setting `errno`.

cpuset_move_all

```
int cpuset_move_all(struct cpuset_pid_list *pl, const char
*cpusetpath);
```

Moves all tasks in list `pl` to cpuset `cpusetpath`.

If the parameter cpusetpath starts with a slash (/) character, this a path relative to `/dev/cpuset`, otherwise, it is relative to the current tasks cpuset.

Returns 0 on success, else -1 on error, setting `errno`.

cpuset_migrate

```
int cpuset_migrate(pid_t pid, const char *cpusetpath);
```

Migrates the task whose process ID is `p` to cpuset `cpusetpath`, moving its currently allocated memory to nodes in that cpuset, if not already there. If `pid` is zero, then the current task is migrated.

If the parameter `cpusetpath` starts with a slash (/) character, this a path relative to `/dev/cpuset`, otherwise, it is relative to the current tasks cpuset.

Returns 0 on success, else -1 on error, setting `errno`.

This is an [optional] function. Use `cpuset_function`() to invoke it.

cpuset_migrate_all

```
int cpuset_migrate_all(struct cpuset_pid_list *pl, const char
*cpusetpath);
```

Moves all tasks in list `pl` to cpuset `cpusetpath`, moving their currently allocated memory to nodes in that cpuset, if not already there.

If the parameter `cpusetpath` starts with a slash (/) character, this a path relative to `/dev/cpuset`, otherwise, it is relative to the current tasks cpuset.

This is an [optional] function. Use `cpuset_function`() to invoke it.

cpuset_reattach

```
int cpuset_reattach(const char *cpusetpath);
```

Reattaches all tasks in cpuset `cpusetpath` to itself. This additional step is necessary anytime that the cpus of a cpuset have been changed, in order to rebind the `cpus_allowed` of each task in the cpuset to the new value. This routine writes the `pid` of each task currently attached to the named cpuset to the tasks file of that cpuset. If additional tasks are being spawned too rapidly into the cpuset at the same time, there is an unavoidable race condition, and some tasks may be missed.

If the parameter `cpusetpath` starts with a slash (/) character, this a path relative to `/dev/cpuset`, otherwise, it is relative to the current tasks cpuset. Returns 0 on success, else -1 on error, setting `errno`.

cpuset_c_rel_to_sys_cpu

```
int cpuset_c_rel_to_sys_cpu(const struct cpuset *cp, int cpu);
```

Returns the system-wide CPU number that is used by the cpu-th CPU of the specified cpuset `cp`. Returns result of `cpuset_cpus_nbits`() if cpu is not in the range [`0, bitmask_weight(cpuset_getcpus(cp))`].

cpuset_c_sys_to_rel_cpu

```
int cpuset_c_sys_to_rel_cpu(const struct cpuset *cp, int cpu);
```

Returns the cpu-th CPU of the specified cpuset `cp` that is used by the system-wide CPU number. Returns result of `cpuset_cpus_nbits`() if `bitmask_isbitset(cpuset_getcpus(cp), cpu)` is false.

cpuset_c_rel_to_sys_mem

```
int cpuset_c_rel_to_sys_mem(const struct cpuset *cp, int mem);
```

Returns the mem-th memory node of the specified cpuset `cp` that is used by the system-wide memory node number. Returns result of `cpuset_mems_nbits`() if `bitmask_isbitset(cpuset_getmems(cp), mem)` is false.

cpuset_c_sys_to_rel_mem

```
int cpuset_c_sys_to_rel_mem(const struct cpuset *cp, int mem);
```

Returns the mem-th memory node of the specified cpuset `cp` that is used by the system-wide memory node number. Returns result of `cpuset_mems_nbits`() if `bitmask_isbitset(cpuset_getmems(cp), mem)` is false.

cpuset_p_rel_to_sys_cpu

```
int cpuset_p_rel_to_sys_cpu(pid_t pid, int cpu);
```

Returns the system-wide CPU number that is used by the cpu-th CPU of the cpuset to which task `pid` is attached. Returns result of `cpuset_cpus_nbits`() if that cpuset does not encompass that relative cpu number.

cpuset_p_sys_to_rel_cpu

```
int cpuset_p_sys_to_rel_cpu(pid_t pid, int cpu);
```

Returns the cpu-th CPU of the cpuset to which task `pid` is attached that is used by the system-wide CPU number. Returns result of `cpuset_cpus_nbits`() if that cpuset does not encompass that system-wide cpu number.

cpuset_p_rel_to_sys_mem

```
int cpuset_p_rel_to_sys_mem(pid_t pid, int mem);
```

Returns the system-wide memory node number that is used by the mem-th memory node of the cpuset to which task `pid` is attached. Returns result of `cpuset_mems_nbits`() if that cpuset does not encompass that relative memory node number.

cpuset_p_sys_to_rel_mem

```
int cpuset_p_sys_to_rel_mem(pid_t pid, int mem);
```

Returns the mem-th memory node of the cpuset to which task `pid` is attached that is used by the system-wide memory node number. Returns result of `cpuset_mems_nbits`() if that cpuset does not encompass that system-wide memory node.

cpuset_cpubind

```
int cpuset_cpubind(int cpu);
```

Binds the scheduling of the current task to CPU `cpu`, using the `sched_setaffinity`(2) system call.

Fails with a return of -1, and `errno` set to `EINVAL`, if `cpu` is not allowed in the current tasks cpuset.

The following code will bind the scheduling of a thread to the n-th CPU of the current cpuset:

```
/*
* Confine current task to only run on the n-th CPU
* of its current cpuset. If in a cpuset of N CPUs,
* valid values for n are 0 .. N-1.
*/
cpuset_cpubind(cpuset_p_rel_to_sys_cpu(0, n));
```

cpuset_latestcpu

```
int cpuset_latestcpu(pid_t pid);
```

Returns the most recent CPU on which the specified task `pid` executed. If `pid` is 0, examine current task.

The `cpuset_latestcpu`() call returns the number of the CPU on which the specified task `pid` most recently executed. If a process can be scheduled on two or more CPUs, the results of `cpuset_lastcpu`() may become invalid even before the query returns to the invoking user code.

The last used CPU is visible for a given `pid` as field #39 (starting with #1) in the file `/proc/pid/stat`. Currently, this file has 41 fields, so it is the 3rd to the last field.

cpuset_membind

```
int cpuset_membind(int mem);
```

Binds the memory allocation of the current task to memory node `mem`, using the `set_mempolicy`(2) system call with a policy of `MPOL_BIND`.

Fails with a return of -1, and `errno` set to `EINVAL`, if `mem` is not allowed in the current tasks cpuset.

The following code will bind the memory allocation of a thread to the n-th memory node of the current cpuset:

```
/*
* Confine current task to only allocate memory on
* n-th Node of its current cpuset.  If in a cpuset
* of N Memory Nodes, valid values for n are 0 .. N-1.
*/
cpuset_membind(cpuset_p_rel_to_sys_mem(0, n));
```

cpuset_export

```
int cpuset_export(const struct cpuset *cp, char *buf, int
buflen);
```

Writes the settings of cpuset `cp` to file. If no file exists at the path specified by file, create one. If a file already exists there, overwrite it.

Returns -1 and sets `errno` on error. Upon successful return, returns the number of characters printed (not including the trailing '0' used to end output to strings). The function `cpuset_export` does not write more than size bytes (including the trailing

'0'). If the output was truncated due to this limit, the return value is the number of characters (not including the trailing '0') which would have been written to the final string if enough space had been available. Thus, a return value of size or more means that the output was truncated.

For details of the format required for exported cpuset file, see "Cpuset Text Format" on page 139.

cpuset_import

```
int cpuset_import(struct cpuset *cp, const char *file, int
*errlinenum_ptr, char *errmsg_bufptr, int errmsg_buflen);
```

Reads the settings of cpuset `cp` from file. If no file exists at the path specified by file, create one. If a file already exists there, overwrite it.

`struct cpuset *cp` must have been returned by a previous `cpuset_alloc`() call. Any previous settings of `cp` are lost.

Returns 0 on success, or -1 on error, setting `errno`. Errors include file not referencing a readable file.

If parsing errors are encountered reading the file and if `errlinenum_ptr` is not NULL, the number of the first line (numbers start with one) with an error is written to `*errlinenum_ptr`. If an error occurs on the open and `errlinenum_ptr` is not NULL, zero is written to `*errlinenum_ptr`.

If parsing errors are encountered reading the file and if `errmsg_bufptr` is not NULL, it is presumed to point to a character buffer of at least `errmsg_buflen` characters and a nul-terminated error message is written to `*errmsg_bufptr`, providing a human readable error message explaining the error message in more detail. Currently, the possible error messages are, as follows:

- "Token 'CPU' requires list"
- "Token 'MEM' requires list"
- "Invalid list format: %s"
- "Unrecognized token: %s"
- "Insufficient memory"

For details of the format required for imported cpuset file, see "Cpuset Text Format" on page 139.

cpuset_function

```
cpuset_function(const char *function_name);
```

Returns pointer to the named `libcpuset.so` function, or NULL. For base functions that are in all implementations of `libcpuset`, there is no particular value in using `cpuset_function`() to obtain a pointer to the function dynamically. But for [optional] cpuset functions, the use of `cpuset_function`() enables dynamically adapting to runtime environments that may or may not support that function.

Application Programming Interface for the Comprehensive System Accounting (CSA)

This appendix contains information about the application programming interface (API) for Comprehensive System Accounting (CSA).

Linux CSA Application Interface Library

The Linux CSA application interface library provides interfaces that allow a programmer access to CSA capabilities, as follows:

Application interface man page	Description
`csa_auth`(3)	Checks to determine if caller has the necessary capabilities.
`csa_check`(3)	Checks a kernel, daemon, or record accounting state.
`csa_halt`(3)	Stops all accounting methods.
`csa_jastart`(3)	Startd job accounting.
`csa_jastop`(3)	Stops job accounting.
`csa_kdstat`(3)	Gets the kernel and daemon accounting status.
`csa_rcdstat`(3)	Gets the record accounting status.
`csa_start`(3)	Gets the user ID of a job.
`csa_stop`(3)	Stops specified accounting method(s).
`csa_wracct`(3)	Writes the accounting record to file.

csa_auth(3)

NAME

csa_auth -checks if caller has the necessary capabilities

LIBRARY

Linux CSA Application Interface library (libcsa_api, -lcsa_api)

SYNOPSIS

```
#include <csa_api.h>
int csa_auth();
```

DESCRIPTION

The csa_auth library call is part of the csa_api library that allows processes to manipulate and obtain status about linux CSA accounting methods. When the csa_api library is to be used, the csa_api.h header file should be included to obtain the proper definitions.

RETURN VALUE

Upon successful verification of proper capabilities, csa_auth returns 0. Otherwise, a value of -1 is returned and errno is set to indicate the error.

ERRORS

Under the following conditions, the csa_auth function fails and sets errno to:

[EACCESS]	Search permission is denied on a component of the path prefix.
[EPERM]	The process does not have appropriate capability to use this system call.
[ENOSYS]	The CSA kernel module/driver is not installed.

SEE ALSO

csa_start(3), csa_stop(3), csa_halt(3), csa_check(3)csa_kdstat(3), csa_rcdstat(3), csa_jastart(3), csa_jastop(3), and csa_wracct(3).

csa_check(3)

NAME

csa_check -checks if caller has the necessary capabilities

LIBRARY

Linux CSA Application Interface library (libcsa_api, -lcsa_api)

SYNOPSIS

```
#include <csa_api.h>

int csa_check( struct csa_check_req *check_req );
```

DESCRIPTION

The csa_check library call is part of the csa_api library that allows processes to manipulate and obtain status about Linux CSA accounting methods. When the csa_api library is to be used, the csa_api.h heade file should be included to obtain the proper definitions.

csa_check checks the state of a specified accounting method.

The caller should provide as a parameter a pointer to a variable of data structure type csa_check_req, as follows:

```
/*
         * CSA_CHECK request and reply
         */
        struct csa_check_req {
            struct csa_am_stat ck_stat;
        };

        /*
         * CSA Accounting Method Status struct
         */
        struct csa_am_stat {
            int     am_id;          /* accounting method ID */
            int     am_status;      /* accounting method status */
            int64_t am_param;       /* accounting method parameter */
        };
```

The state of the inquired accounting method is returned in am_status.

RETURN VALUE

Upon successful completion, `csa_check` returns 0 and `check_req->ck_status` contains the status of the inquired accounting method. Otherwise, a value of -1 is returned and `errno` is set to indicate the error.

ERRORS

Under the following conditions, the `csa_check` function fails and sets `errno` to:

`[EACCESS]`	Search permission is denied on a component of the path prefix.
`[EFAULT]`	The `check_req argument` points to an illegal address.
`[EINVAL]`	An invalid argument was specified.
`[EPERM]`	The process does not have appropriate capability to use this system call.
`[ENOSYS]`	The CSA kernel module/driver is not installed.
`[ENOENT]`	No job table entry is found when attempting to start or stop user job accounting.

SEE ALSO

`csa_start`(3), `csa_stop`(3), `csa_halt`(3), `csa_kdstat`(3), `csa_rcdstat`(3), `csa_jastart`(3), `csa_jastop`(3) `csa_wracct`(3), and `csa_auth`(3).

csa_halt(3)

NAME

csa_halt -stops all accounting methods

LIBRARY

Linux CSA Application Interface library (libcsa_api, -lcsa_api)

SYNOPSIS

```
#include <csa_api.h>

int csa_halt();
```

DESCRIPTION

The csa_halt library call is part of the csa_api library that allows processes to manipulate and obtain status about Linux CSA accounting methods. When the csa_api library is to be used, the csa_api.h header file should be included to obtain the proper definitions.

csa_halt stops all accounting methods.

RETURN VALUE

Upon successful verification of proper capabilities, csa_halt returns 0. Otherwise, a value of -1 is returned and errno is set to indicate the error.

ERRORS

Under the following conditions, the csa_halt function fails and sets errno to:

[EACCESS]	Search permission is denied on a component of the path prefix.
[EPERM]	The process does not have appropriate capability to use this system call.
[ENOSYS]	The CSA kernel module/driver is not installed.
[ENOENT]	No job table entry is found when attempting to start or stop user job accounting.

SEE ALSO

`csa_start`(3), `csa_stop`(3), `csa_check`(3)`csa_kdstat`(3), `csa_rcdstat`(3), `csa_jastart`(3), `csa_jastop`(3), `csa_auth`(3) and `csa_wracct`(3).

csa_jastart(3)

NAME

csa_jastart -starts job accounting

LIBRARY

Linux CSA Application Interface library (libcsa_api, -lcsa_api)

SYNOPSIS

```
#include <csa_api.h>

int csa_jastart( struct csa_job_req *job_req );
```

The csa_jastart library call is part of the csa_api library that allows processes to manipulate and obtain status about Linux CSA accounting methods. When the csa_api library is to be used, the csa_api.h header file should be included to obtain the proper definitions.

csa_jastart starts job accounting.

The caller should provide as a parameter a pointer to a variable of data structure type csa_job_req, as follows:

```
/*
         * CSA_JASTART/CSA_JASTOP request
         */
        struct csa_job_req {
            char    job_path[ACCT_PATH+1];
        };
```

RETURN VALUE

Upon successful completion, csa_jastart returns 0. Otherwise, a value of -1 is returned and errno is set to indicate the error.

ERRORS

Under the following conditions, the csa_jastart function fails and sets errno to:

[EACCESS]	Search permission is denied on a component of the path prefix.
[EFAULT]	The check_req argument points to an illegal address.

[EINVAL]	An invalid argument was specified.
[EPERM]	The process does not have appropriate capability to use this system call.
[ENOSYS]	The CSA kernel module/driver is not installed.
[ENOENT]	No job table entry is found when attempting to start or stop user job accounting.

SEE ALSO

csa_start(3), csa_stop(3), csa_halt(3), csa_check(3),csa_kdstat(3), csa_rcdstat(3), csa_jastop(3) csa_wracct(3), and csa_auth(3).

csa_jastop(3)

NAME

csa_jastop -stops job accounting

LIBRARY

Linux CSA Application Interface library (libcsa_api, -lcsa_api)

SYNOPSIS

```
#include <csa_api.h>

int csa_jastop( struct csa_job_req *job_req );
```

DESCRIPTION

The csa_jastop library call is part of the csa_api library that allows processes to manipulate and obtain status about Linux CSA accounting methods. When the csa_api library is to be used, the csa_api.h heade file should be included to obtain the proper definitions.

csa_jastop stops job accounting.

The caller should provide as a parameter a pointer to a variable of data structure type csa_job_req, as follows:

```
/*
         * CSA_JASTART/CSA_JASTOP request
         */
        struct csa_job_req {
            char    job_path[ACCT_PATH+1];
        };
```

RETURN VALUE

Upon successful completion, csa_jastop returns 0. Otherwise, a value of -1 is returned and errno is set to indicate the error.

ERRORS

Under the following conditions, the csa_jastop function fails and sets errno to:

[EACCESS]	Search permission is denied on a component of the path prefix.

[EFAULT]	The `check_req argument` points to an illegal address.
[EINVAL]	An invalid argument was specified.
[EPERM]	The process does not have appropriate capability to use this system call.
[ENOSYS]	The CSA kernel module/driver is not installed.
[ENOENT]	No job table entry is found when attempting to start or stop user job accounting.

SEE ALSO

`csa_start`(3), `csa_stop`(3), `csa_halt`(3), `csa_check`(3),`csa_kdstat`(3), `csa_rcdstat`(3), `csa_jastart`(3) `csa_wracct`(3), and `csa_auth`(3).

csa_kdstat(3)

NAME

csa_kdstat -gets the kernel and daemon accounting status

LIBRARY

Linux CSA Application Interface library (libcsa_api, -lcsa_api)

SYNOPSIS

```
#include <csa_api.h>

int csa_kdstat( struct csa_status_req *kdstat_req );
```

DESCRIPTION

The csa_kdstat library call is part of the csa_api library that allows processes to manipulate and obtain status about Linux CSA accounting methods. When the csa_api library is to be used, the csa_api.h heade file should be included to obtain the proper definitions.

csa_kdstat gets the kernel and daemon accounting status.

The caller should provide as a parameter a pointer to a variable of data structure type csa_status_req, as follows:

```
/*
         * CSA_KDSTAT/CSA_RCDSTAT request
         */
        struct csa_status_req {
            int     st_num;         /* num of entries in kd_method array */
            char    st_path[ACCT_PATH+1];
            struct csa_am_stat st_stat[NUM_KDRCDS];
        };

        /*
         * CSA Accounting Method Status struct
         */
        struct csa_am_stat {
            int     am_id;          /* accounting method ID */
            int     am_status;      /* accounting method status */
            int64_t am_param;       /* accounting method parameter */
```

```
};
```

The inquired status are returned in the `am_status` field of `st_stat` array.

RETURN VALUE

Upon successful completion, `csa_kdstat` returns 0. Otherwise, a value of -1 is returned and `errno` is set to indicate the error.

ERRORS

Under the following conditions, the `csa_kdstat` function fails and sets `errno` to:

`[EACCESS]`	Search permission is denied on a component of the path prefix.
`[EFAULT]`	The `check_req argument` points to an illegal address.
`[EINVAL]`	An invalid argument was specified.
`[EPERM]`	The process does not have appropriate capability to use this system call.
`[ENOSYS]`	The CSA kernel module/driver is not installed.
`[ENOENT]`	No job table entry is found when attempting to start or stop user job accounting.

SEE ALSO

`csa_start`(3), `csa_stop`(3), `csa_halt`(3), `csa_check`(3),`csa_jastop`(3), `csa_rcdstat`(3), `csa_jastart`(3) `csa_wracct`(3), and `csa_auth`(3).

csa_rcdstat(3)

NAME

csa_rcdstat -gets the record accounting status

LIBRARY

Linux CSA Application Interface library (libcsa_api, -lcsa_api)

SYNOPSIS

```
#include <csa_api.h>

int csa_rcdstat( struct csa_status_req *rcdstat_req );
```

DESCRIPTION

The csa_rcdstat library call is part of the csa_api library that allows processes to manipulate and obtain status about Linux CSA accounting methods. When the csa_api library is to be used, the csa_api.h header file should be included to obtain the proper definitions.

csa_rcdstat gets the record accounting status.

The caller should provide as a parameter a pointer to a variable of data structure type csa_status_req, as follows:

```
/*
         * CSA_KDSTAT/CSA_RCDSTAT request
         */
        struct csa_status_req {
            int     st_num;         /* num of entries in kd_method array */
            char    st_path[ACCT_PATH+1];
            struct csa_am_stat st_stat[NUM_KDRCDS];
        };

        /*
         * CSA Accounting Method Status struct
         */
        struct csa_am_stat {
            int     am_id;          /* accounting method ID */
            int     am_status;      /* accounting method status */
            int64_t am_param;       /* accounting method parameter */
        };
```

The inquired status are returned in the `am_status` field of `st_stat` array.

RETURN VALUE

Upon successful verification of proper capabilities, `csa_rcdstat` returns 0. Otherwise, a value of -1 is returned and `errno` is set to indicate the error.

ERRORS

Under the following conditions, the `csa_rcdstat` function fails and sets `errno` to:

`[EACCESS]`	Search permission is denied on a component of the path prefix.
`[EFAULT]`	The `check_req argument` points to an illegal address.
`[EINVAL]`	An invalid argument was specified.
`[EPERM]`	The process does not have appropriate capability to use this system call.
`[ENOSYS]`	The CSA kernel module/driver is not installed.
`[ENOENT]`	No job table entry is found when attempting to start or stop user job accounting.

SEE ALSO

`csa_start`(3), `csa_stop`(3), `csa_halt`(3), `csa_check`(3),`csa_jastop`(3), `csa_kdstat`(3), `csa_jastart`(3) `csa_wracct`(3), and `csa_auth`(3).

csa_start(3)

NAME

csa_start -starts specified accounting method(s)

LIBRARY

Linux CSA Application Interface library (libcsa_api, -lcsa_api)

SYNOPSIS

```
#include <csa_api.h>
```

DESCRIPTION

The csa_start library call is part of the csa_api library that allows processes to manipulate and obtain status about Linux CSA accounting methods. When the csa_api library is to be used, the csa_api.h header file should be included to obtain the proper definitions.

csa_start starts specified CSA accounting method(s).

The caller should provide as a parameter a pointer to a variable of data structure type struct csa_start_req, as follows:

```
/*
         * CSA_START request
         */
        struct csa_start_req {
            int    sr_num;         /* num of entries in sr_method array */
            char   sr_path[ACCT_PATH+1];  /* path name for accounting file */
            struct method_info {
                   int     sr_id;         /* Accounting Method type id */
                   int64_t param;         /* Entry parameter */
            } sr_method[NUM_KDRCDS];
        }
```

RETURN VALUE

Upon successful verification of proper capabilities, csa_start returns 0. Otherwise, a value of -1 is returned and errno is set to indicate the error.

ERRORS

Under the following conditions, the `csa_start` function fails and sets `errno` to:

`[EACCESS]`	Search permission is denied on a component of the path prefix.
`[EFAULT]`	The `check_req argument` points to an illegal address.
`[EINVAL]`	An invalid argument was specified.
`[EPERM]`	The process does not have appropriate capability to use this system call.
`[ENOSYS]`	The CSA kernel module/driver is not installed.
`[ENOENT]`	No job table entry is found when attempting to start or stop user job accounting.

SEE ALSO

`csa_stop`(3), `csa_halt`(3), `csa_check`(3),`csa_jastop`(3), `csa_kdstat`(3), `csa_rcdstat`(3),`csa_jastart`(3) `csa_wracct`(3), and `csa_auth`(3).

csa_stop(3)

NAME

csa_stop -

LIBRARY

Linux CSA Application Interface library (libcsa_api, -lcsa_api)

SYNOPSIS

```
#include <csa_api.h>

int csa_stop( struct csa_stop_req *stop_req );
```

DESCRIPTION

The csa_stop library call is part of the csa_api library that allows processes to manipulate and obtain status about Linux CSA accounting methods. When the csa_api library is to be used, the csa_api.h header file should be included to obtain the proper definitions.

csa_stops stops specified CSA accounting method(s).

The caller should provide as a parameter a pointer to a variable of data structure type csa_stop_req, as follows:

```
/*
         * CSA_STOP request
         */
        struct csa_stop_req {
            int     pr_num;         /* num of entries in pr_id[] array */
            int     pr_id[NUM_KDRCDS];   /* Accounting Method type id */
        };
```

RETURN VALUE

Upon successful verification of proper capabilities, csa_stop returns 0. Otherwise, a value of -1 is returned and errno is set to indicate the error.

ERRORS

Under the following conditions, the csa_stop function fails and sets errno to:

[EACCESS]	Search permission is denied on a component of the path prefix.
[EFAULT]	The `check_req argument` points to an illegal address.
[EINVAL]	An invalid argument was specified.
[EPERM]	The process does not have appropriate capability to use this system call.
[ENOSYS]	The CSA kernel module/driver is not installed.
[ENOENT]	No job table entry is found when attempting to start or stop user job accounting.

SEE ALSO

`csa_start`(3), `csa_halt`(3), `csa_check`(3),`csa_jastop`(3), `csa_kdstat`(3), `csa_rcdstat`(3),`csa_jastart`(3) `csa_wracct`(3), and `csa_auth`(3).

csa_wracct(3)

NAME

csa_wracct -writes an accounting record to a file

LIBRARY

Linux CSA Application Interface library (libcsa_api, -lcsa_api)

SYNOPSIS

```
#include <csa_api.h>

int csa_wracct( struct csa_wra_req *wra_req );
```

DESCRIPTION

The csa_wracct library call is part of the csa_api library that allows processes to manipulate and obtain status about Linux CSA accounting methods. When the csa_api library is to be used, the csa_api.h header file should be included to obtain the proper definitions.

csa_wracct writes an accounting record to a file.

The caller should provide as a parameter a pointer to a variable of data structure type csa_wra_req, as follows:

```
/*
         * CSA_WRACCT request
         */
        struct csa_wra_req {
            int             wra_did;        /* Daemon Index */
            int             wra_len;        /* Length of buffer (bytes) */
            uint64_t        wra_jid;        /* Job ID */
            char            *wra_buf;       /* Daemon accounting buffer */
        };
```

RETURN VALUE

Upon successful verification of proper capabilities, csa_wracct returns 0. Otherwise, a value of -1 is returned and errno is set to indicate the error.

ERRORS

Under the following conditions, the `csa_wracct` function fails and sets `errno` to:

`[EACCESS]`	Search permission is denied on a component of the path prefix.
`[EFAULT]`	The `check_req argument` points to an illegal address.
`[EINVAL]`	An invalid argument was specified.
`[EPERM]`	The process does not have appropriate capability to use this system call.
`[ENOSYS]`	The CSA kernel module/driver is not installed.
`[ENOENT]`	No job table entry is found when attempting to start or stop user job accounting.

SEE ALSO

`csa_start`(3), `csa_stop`(3)`csa_halt`(3), `csa_check`(3),`csa_jastop`(3), `csa_kdstat`(3), `csa_rcdstat`(3),`csa_jastart`(3), and `csa_auth`(3).

Index