Linux® Resource Administration Guide

007–4413–001

CONTRIBUTORS

Written by Terry Schultz

Edited by Susan Wilkening

Illustrated by Chris Wengelski

Production by Glen Traefald

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

Cover Design By Sarah Bolles, Sarah Bolles Design, and Dany Galgani, SGI Technical Publications.

Record of Revision

Version	Description
001	February 2003 Original publication.

Contents

Figures

Tables

About This Guide

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

This manual contains the following chapters:

Related Publications

For a list of Comprehensive System Accounting (CSA) man pages, see "CSA Man Pages", page 52.

For a list of Array Services man pages, see "Using Array Services Commands", page 62.

Obtaining Publications

You can obtain SGI documentation in the following ways:

- See the SGI Technical Publications Library at: `http://docs.sgi.com`. Various formats are available. This library contains the most recent and most comprehensive set of online books, release notes, man pages, and other information.
- SGI ProPack for Linux documentation, and all other documentation included in the RPMs on the distribution CDs can be found on the CD titled "SGI ProPack V.2.1 for Linux - Documentation CD." To access the information on the documentation

CD, open the index.html file with a web browser. Because this online file can be updated later in the release cycle than this document, you should check it for the latest information. After installation, all SGI ProPack for Linux documentation (including `README.SGI`) is in `/usr/share/doc/sgi-propack-2.1`.

- 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 document, contact SGI. Be sure to include the title and document number of the manual with your comments. (Online, the document number is located in the front matter of the manual. In printed manuals, 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
  1600 Amphitheatre Parkway, M/S 535
  Mountain View, California 94043–1351

- Send a fax to the attention of "Technical Publications" at +1 650 932 0801.

SGI values your comments and will respond to them promptly.

Linux Kernel Jobs

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

- "Overview", page 1
- "Installing and Configuring Linux Kernel Jobs", page 3

Overview

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

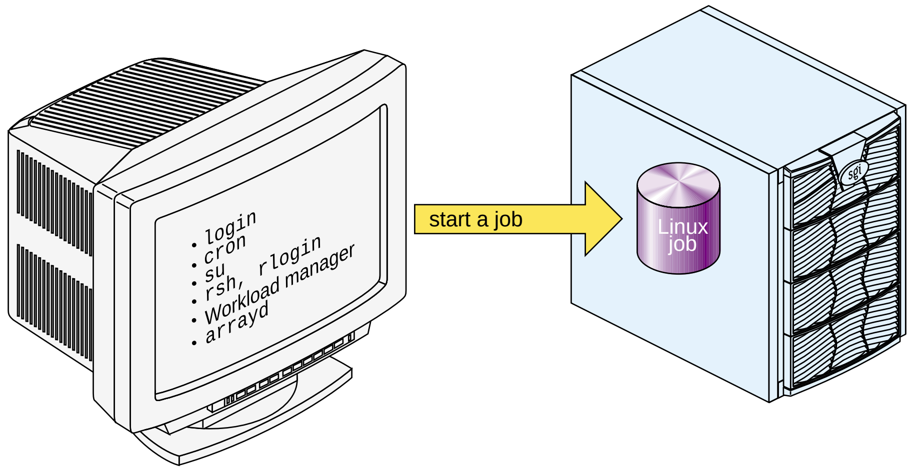

Figure 1-1 Point-of-Entry Processes

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

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

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

Installing and Configuring Linux Kernel Jobs

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

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

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

2. Enable job creation for `login` users by adding this entry to the `login` configuration file:

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

   This example shows the `login` configuration file being changed. You need to add the `session` line to all of the PAM entry points that will create jobs on your system, for example, `login`, `rlogin`, `rsh`, `su`, and `xdm`.

3. To configure jobs on across system reboots, use the `chkconfig`(8) command as follows:

   ```
   chkconfig --add job
   ```

4. To stop jobs from initiating after a system reboot, 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 is dependent on the concept of a Linux kernel job. For more information on Linux kernel jobs, see Chapter 1, "Linux Kernel Jobs", page 1.

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.

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)", page 38.

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.

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", page 9.

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", page 3.

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

   ```
   chkconfig --add csaacct
   ```

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/csaacct 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", page 19.

The following steps are required to disable CSA job accounting:

1. To turn off CSA, enter the following:

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

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

   ```
   chkconfig --del csaacct
   ```

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:

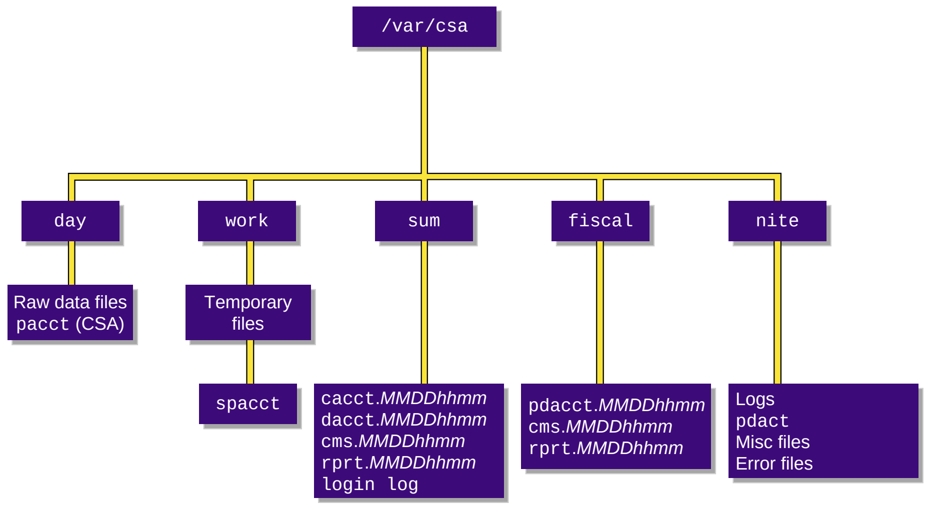

Figure 2-1 The `/var/csa` Directory

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

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", page 19).
`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", page 19).
`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", page 19).

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` 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", page 43. 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", page 19.
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", page 38. 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", page 26.

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 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 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.

CSA system diagram

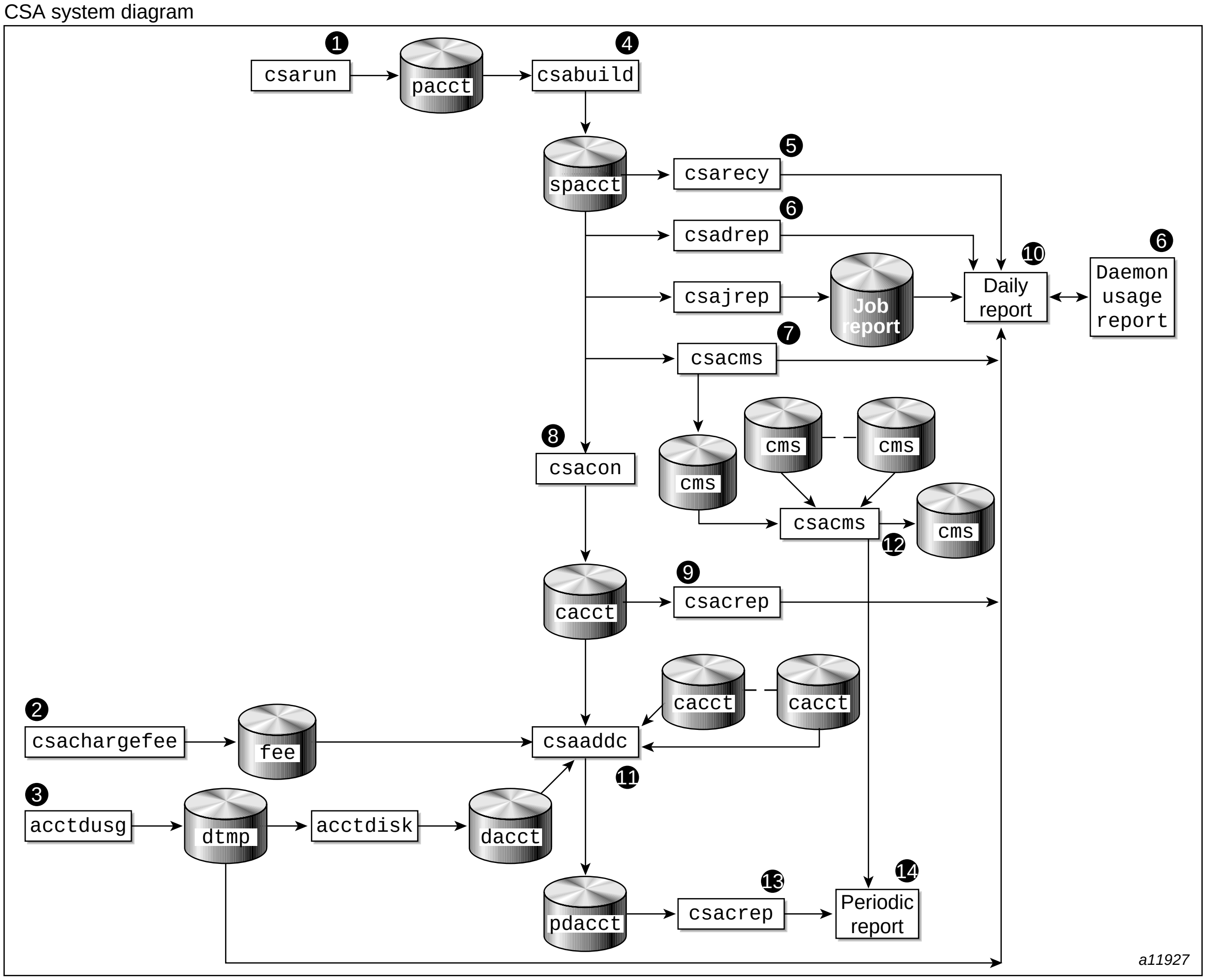

Figure 2-2 CSA Data Processing

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", page 32).

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", page 19, 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", page 33.

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", page 35. 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", page 37.
- 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", page 33.

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", page 41).
- 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", page 19, 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", page 19). 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.
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.
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", page 43.

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", page 47
- "Unfinished Job Information Report", page 48
- "Disk Usage Report", page 48
- "Command Summary Report", page 48
- "Last Login Report", page 49
- "Daemon Usage Report", page 49

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", page 51
- "Command summary report", page 51

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*.

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.

Chapter 3

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", page 58.

Installing and Configuring Array Services

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

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

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

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

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

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

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

```
chkconfig --add array
```

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

   - "About Array Configuration", page 74
   - "Configuring Arrays and Machines", page 79
   - "Configuring Authentication Codes", page 80
   - "Configuring Array Commands", page 81

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", page 58, summarizes what a user needs to know and the main facilities a user has available.
- "Managing Local Processes", page 61, reviews the conventional tools for listing and controlling processes within one node.
- "Using Array Services Commands", page 62, describes the common concepts, options, and environment variables used by the Array Services commands.
- "Interrogating the Array", page 66, summarizes how to use Array Services commands to learn about the Array and its workload, with examples.
- "Summary of Common Command Options", page 64
- "Managing Distributed Processes", page 69, 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, page 61.

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, page 62, 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, page 62.

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.
`-F`	`ainfo`, `array`, `arshell, aview`	Forward commands to other nodes through the `array` daemon.

Option	Used In	Description
-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 guest      1.19 atop
0x1111ffff000001e9 tokyo        19816 guest      0.73 atop
0x1111ffff000001e9 disarray      1106 guest      0.47 atop
0x1111ffff000001e9 datarray      1423 guest      0.42 atop
0x1111ffff00000000 homegrown       20 root       0.41 ShareII
0x1111ffff000000c0 homegrown    29683 kchang     0.37 ld
0x1111ffff0000001e homegrown     1324 root       0.17 arrayd
0x1111ffff00000000 homegrown      229 root       0.14 routed
0x1111ffff00000000 homegrown       19 root       0.09 pdflush
0x1111ffff000001e9 disarray      1105 guest      0.02 atopm
```

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

Managing Distributed Processes

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

About Array Session Handles (ASH)

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

An ASH is a number that is unique across the entire array (unlike a PID or PGID). An ASH is the same for every process that is part of a single array session—no matter which node the process runs in. You display and use ASH values with Array Services commands. Each time you log in to an Array node, your shell is given an ASH, which is used by all the processes you start from that shell.

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

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

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

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

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

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

Listing Processes and ASH Values

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

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

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

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

Controlling Processes

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

Using arshell

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

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

You can use `arshell` to start a collection of unrelated programs in multiple nodes under a single ASH; then you can use the commands described under "Managing Session Processes", page 73 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", page 74.)

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

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

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

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

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

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

Managing Session Processes

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

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

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

```
homegrown 176% array spin
```

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

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

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

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

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

About Job Container IDs

Array systems have the capability to forward job IDs (JIDs) from the initiating host. All of the processes running in the ASH across one or more nodes in an array also belong to the same job. For a complete description of the job container and it usage, see Chapter 1, "Linux Kernel Jobs", 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", page 64).
- 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", page 85.)

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", page 64). 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", page 84.

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", page 64).

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", page 71). 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 “guest.”

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

Designing New Array Commands

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

Typically, a new command will be defined with an INVOKE subentry that names a script written in sh, csh, or Perl syntax. You use the substitution values to set up arguments to the script. You use the USER, GROUP, PROJECT, and OPTIONS subentries to establish the execution conditions of the script. For one example of a command definition using a simple script, see "About the Distributed Example", page 72.

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

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

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

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

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

The command definition would be as follows:

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

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

CPU Memory Sets and Scheduling

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

CpuMemSets provides a Linux kernel facility that enables system services and applications to specify on which CPUs they may be scheduled and from which nodes they may allocate memory. 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", page 94. For more information on the Python interface, see "Managing CpuMemSets", page 95.

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. 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", page 90

• "cpumemset", page 90

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", page 92
- "Configuring CpuMemSets", page 93
- "Tuning CpuMemSets", page 93

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", page 94
- "Initializing CpuMemSets", page 94
- "Operating on CpuMemSets", page 95
- "Managing CpuMemSets", page 95
- "Initializing System Service on CpuMemSets", page 96
- "Resolving Pages for Memory Areas", page 96
- "Determining an Application’s Current CPU", page 97
- "Determining the Memory Layout of cpumemmaps and cpumemsets", page 97

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", page 97).

Managing CpuMemSets

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

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

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

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

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

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

Initializing System Service on CpuMemSets

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

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

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

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

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

Resolving Pages for Memory Areas

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

Determining an Application's Current CPU

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

Determining the Memory Layout of cpumemmaps and cpumemsets

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

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

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

Hard Partitioning versus CpuMemSets

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

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

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

Error Messages

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

- If a request is made to set a cpumemmap that has fewer CPUs or memory blocks listed than needed by any cpumemsets that will be using that cpumemmap after the change, the cmsSetCMM() call fails, with errno set to ENOENT. You cannot remove elements of a cpumemmap that are in use.
- If a request is made to set a cpumemset that references CPU or memory blocks not available in its current cpumemmap, the cmsSetCMS() call fails, with errno set to ENOENT. You cannot reference unmapped application CPUs or memory blocks in a cpumemset.
- If a request is made without root privileges to set a cpumemmap by a process , and that request attempts to add any system CPU or memory block number not currently in the map being changed, the request fails, with errno set to EPERM.
- If a cmsSetCMS() request is made on another process, the requesting process must either have root privileges, or the real or effective user ID of the sending process must equal the real or saved set-user-ID of the other process, or else the request fails, with errno set to EPERM. These permissions are similar to those required by the kill(2) system call.
- Every cpumemset must specify a memory list for the CMS_DEFAULT_CPU, to ensure that regardless of which CPU a memory request is executed on, a memory list will be available to search for memory. Attempts to set a cpumemset without a memory list specified for the CMS_DEFAULT_CPU fail, with errno set to EINVAL.
- If a request is made to set a cpumemset that has the same CPU (application number) listed in more than one array cpus of CPUs sharing any cms_memory_list_t structures, then the request fails, with errno set to EINVAL. Otherwise, duplicate CPU or memory block numbers are harmless, except for minor inefficiencies.
- The operations to query and set cpumemmaps and cpumemsets can be applied to any process ID (PID). If the PID is zero, then the operation is applied to the current process. If the specified PID does not exist, then the operation fails, with errno set to ESRCH.

Cpuset System

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

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

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

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

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

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

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

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

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

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

This chapter contains the following sections:

Cpusets on Linux versus IRIX

This sections describes the major differences between how the Cpuset System is implemented on the Linux operating system for the SGI Linux Environment 7.2 release versus the current IRIX operating system. These differences are likely to change for future releases of the SGI Linux Environment.

Major differences include the following:

- Linux does not have the explicit concept of a boot cpuset. The boot cpuset is implicit on Linux systems. All processes run on the entire system and can use any system memory unless otherwise placed on a cpuset. For an example of how to create a “virtual” boot cpuset on your SGI Linux system, see Example 5-2, page 107.
- 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 release. For more information, see "Cpuset Configuration File", page 107.
- 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 include only 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.
- 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", page 107.
- 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", page 87.
- 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.
- The current Linux release does not support the cpuset library routines`cpusetAttachPid`(3x) and `cpusetDetachPid`(3x), which can be used to attach or detach a specific process from a cpuset.
- 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.

Using Cpusets

This section describes the basic steps for using cpusets and the `cpuset`(1) command. For a detailed example, see "Cpuset System Examples", page 104.

To install the Cpuset System software, see "Installing the Cpuset System", page 110.

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", page 107. For restrictions that apply to CPUs belonging to cpusets, see "Restrictions on CPUs within Cpusets", page 104.
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*`] [-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.

`-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", page 104, and "Cpuset System Examples", page 104.

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", page 107.

   For an explanation of the `MEMORY_LOCAL` flag, see "Cpuset Configuration File", page 107.

   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", page 107.

   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.

Example 5-2 Creating a "Boot" Cpuset

You can create a "boot" cpuset 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 as follows:

1. To constrain your system, including the kernel, user logins, and all processes to just one CPU and one node, before the init process begins executing, set the following kernel boot option (accessible via elilo)

   ```
   cpumemset_minimal=1
   ```

   For more information on kernel boot command line options, see "cpumemset", page 90 and "Tuning CpuMemSets", page 93.

2. To configure the rest of your system, follow the steps in Example 5-1 to create cpusets and assign specific applications to execute on them. The system resources, other than the one CPU and the one node running init, the kernel, and all processes, remain "dark" until explicitly attached to a cpuset with one exception as follows:

   If there is no free memory on the current node when an application requests memory, memory may be acquired from other nodes, which may or may not be in the cpuset or CpuMemSet specified for that process. This behavior is subject to change in future releases of SGI Linux.

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

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 **Not implemented on Linux**
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**

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

Miscellaneous Man Pages

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

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

NUMA Tools

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

This chapter covers the following topics:

dlook

The `dlook`(1) command allows you to display the memory map and CPU usage for a specified process as follows:

`dlook [-a] [-c] [-h] [-l] [-o` *outfile*`] [-s` *secs*`] command [`*command-args*`]`
`dlook [-a] [-c] [-h] [-l] [-o` *outfile*`] [-s secs]` *pid*

For each page in the virtual address space of the process, `dlook`(1) prints the following information:

- The object that owns the page, such as a file, SYSV shared memory, a device driver, and so on.
- The type of page, such as random access memory (RAM), FETCHOP, IOSPACE, and so on.
- If the page type is RAM memory, the following information is displayed:
  - Memory attributes, such as, SHARED, DIRTY, and so on

  - The node on which the page is located
  - The physical address of the page (optional)

- Optionally, the `dlook`(1) command also prints the amount of elapsed CPU time that the process has executed on each physical CPU in the system.

Two forms of the `dlook`(1) command are provided. In one form, `dlook` prints information about an existing process that is identified by a process ID (PID). To use this form of the command, you must be the owner of the process or be running with root privilege. In the other form, you use `dlook` on a command you are launching and thus are the owner.

The `dlook`(1) command accepts the following options:

`-a`	Shows the physical addresses of each page in the address space.
`-c`	Shows the elapsed CPU time, that is how long the process has executed on each CPU.
`-h`	Explicitly lists holes in the address space.
`-l`	Shows libraries.
`-o`	Outputs the file name. If not specified, output is written to stdout.
`-s`	Specifies a sample interval in seconds. Information about the process is displayed every second (`secs`) of CPU usage by the process.

An example for the `sleep` process with a PID of 4702 is as follows:

Note: The output has been abbreviated to shorten the example and bold headings added for easier reading.

```
dlook 4702

Peek:  sleep
Pid: 4702       Thu Aug 22 10:45:34 2002

Cputime by cpu (in seconds):
                  user    system
  TOTAL          0.002     0.033
  cpu1           0.002     0.033

Process memory map:
  2000000000000000-2000000000030000 r-xp 0000000000000000 04:03 4479 /lib/ld-2.2.4.so
```

```
        [2000000000000000-200000000002c000]        11 pages on node   1  MEMORY|SHARED

  2000000000030000-200000000003c000 rw-p 0000000000000000 00:00 0
        [2000000000030000-200000000003c000]         3 pages on node   0  MEMORY|DIRTY

                                     ...

  2000000000128000-2000000000370000 r-xp 0000000000000000 04:03 4672       /lib/libc-2.2.4.so
        [2000000000128000-2000000000164000]        15 pages on node   1  MEMORY|SHARED
        [2000000000174000-2000000000188000]         5 pages on node   2  MEMORY|SHARED
        [2000000000188000-2000000000190000]         2 pages on node   1  MEMORY|SHARED
        [200000000019c000-20000000001a8000]         3 pages on node   1  MEMORY|SHARED
        [20000000001c8000-20000000001d0000]         2 pages on node   1  MEMORY|SHARED
        [20000000001fc000-2000000000204000]         2 pages on node   1  MEMORY|SHARED
        [200000000020c000-2000000000230000]         9 pages on node   1  MEMORY|SHARED
        [200000000026c000-2000000000270000]         1 page  on node   1  MEMORY|SHARED
        [2000000000284000-2000000000288000]         1 page  on node   1  MEMORY|SHARED
        [20000000002b4000-20000000002b8000]         1 page  on node   1  MEMORY|SHARED
        [20000000002c4000-20000000002c8000]         1 page  on node   1  MEMORY|SHARED
        [20000000002d0000-20000000002d8000]         2 pages on node   1  MEMORY|SHARED
        [20000000002dc000-20000000002e0000]         1 page  on node   1  MEMORY|SHARED
        [2000000000340000-2000000000344000]         1 page  on node   1  MEMORY|SHARED
        [200000000034c000-2000000000358000]         3 pages on node   2  MEMORY|SHARED

                                          ....

  20000000003c8000-20000000003d0000 rw-p 0000000000000000 00:00 0
        [20000000003c8000-20000000003d0000]         2 pages on node   0  MEMORY|DIRTY
```

The `dlook` command gives the name of the process (`Peek:   sleep`), the process ID, and time and date it was invoked. It provides total user and system CPU time in seconds for the process.

Under the heading **Process memory map**, the `dlook` command prints information about a process from the `/proc/`*pid*`/cpu` and `/proc/`*pid*`/maps` files. On the left, it shows the memory segment with the offsets below in decimal. In the middle of the output page, it shows the type of access, time of execution, the PID, and the object that owns the memory (in this case, `/lib/ld-2.2.4.so`). The characters `s` or `p` indicate whether the page is mapped as sharable (`s`) with other processes or is private (`p`). The right side of the output page shows the number of pages of memory consumed and on which nodes the pages reside. *Dirty memory* means that the memory has been modified by a user.

In the second form of the `dlook` command, you specify a command and optional command arguments. The `dlook` command issues an `exec` call on the command and passes the command arguments. When the process terminates, `dlook` prints information about the process, as shown in the following example:

`dlook` *date*

```
Thu Aug 22 10:39:20 CDT 2002
_______________________________________________________________________________
Exit:  date
Pid: 4680       Thu Aug 22 10:39:20 2002


Process memory map:
  2000000000030000-200000000003c000 rw-p 0000000000000000 00:00 0
        [2000000000030000-200000000003c000]         3 pages on node   3  MEMORY|DIRTY

  20000000002dc000-20000000002e4000 rw-p 0000000000000000 00:00 0
        [20000000002dc000-20000000002e4000]         2 pages on node   3  MEMORY|DIRTY

  2000000000324000-2000000000334000 rw-p 0000000000000000 00:00 0
        [2000000000324000-2000000000328000]         1 page  on node   3  MEMORY|DIRTY

  4000000000000000-400000000000c000 r-xp 0000000000000000 04:03 9657220    /bin/date
        [4000000000000000-400000000000c000]         3 pages on node   1  MEMORY|SHARED

  6000000000008000-6000000000010000 rw-p 0000000000008000 04:03 9657220    /bin/date
        [600000000000c000-6000000000010000]         1 page  on node   3  MEMORY|DIRTY

  6000000000010000-6000000000014000 rwxp 0000000000000000 00:00 0
        [6000000000010000-6000000000014000]         1 page  on node   3  MEMORY|DIRTY

  60000fff80000000-60000fff80004000 rw-p 0000000000000000 00:00 0
        [60000fff80000000-60000fff80004000]         1 page  on node   3  MEMORY|DIRTY

  60000ffffff4000-60000fffffffc000 rwxp ffffffffffffc000 00:00 0
        [60000ffffff4000-60000fffffffc000]         2 pages on node   3  MEMORY|DIRTY
```

If you use the `dlook` command with the `-s` *secs* option, the information is sampled at regular internals. The output for the command **`dlook -s 5 sleep 50`** is as follows:

```
Exit:  sleep
Pid: 5617       Thu Aug 22 11:16:05 2002


Process memory map:
  2000000000030000-200000000003c000 rw-p 0000000000000000 00:00 0
        [2000000000030000-200000000003c000]            3 pages on node   3  MEMORY|DIRTY

  2000000000134000-2000000000140000 rw-p 0000000000000000 00:00 0

  20000000003a4000-20000000003a8000 rw-p 0000000000000000 00:00 0
        [20000000003a4000-20000000003a8000]            1 page  on node   3  MEMORY|DIRTY

  20000000003e0000-20000000003ec000 rw-p 0000000000000000 00:00 0
        [20000000003e0000-20000000003ec000]            3 pages on node   3  MEMORY|DIRTY

  4000000000000000-4000000000008000 r-xp 0000000000000000 04:03 9657225    /bin/sleep
        [4000000000000000-4000000000008000]            2 pages on node   3  MEMORY|SHARED

  6000000000004000-6000000000008000 rw-p 0000000000004000 04:03 9657225    /bin/sleep
        [6000000000004000-6000000000008000]            1 page  on node   3  MEMORY|DIRTY

  6000000000008000-600000000000c000 rwxp 0000000000000000 00:00 0
        [6000000000008000-600000000000c000]            1 page  on node   3  MEMORY|DIRTY

  60000fff80000000-60000fff80004000 rw-p 0000000000000000 00:00 0
        [60000fff80000000-60000fff80004000]            1 page  on node   3  MEMORY|DIRTY

  60000ffffff4000-60000fffffffc000 rwxp ffffffffffffc000 00:00 0
        [60000ffffff4000-60000fffffffc000]            2 pages on node   3  MEMORY|DIRTY
```

You can run an message passing interface (MPI) job using the `mpirun` command and print the memory map for each thread, or redirect the ouput to a file, as follows:

Note: The output has been abbreviated to shorten the example and bold headings added for easier reading.

```
mpirun -np 8 dlook -o dlook.out ft.C.8

Contents of dlook.out:
_______________________________________________________________________________
Exit:  ft.C.8
Pid: 2306       Fri Aug 30 14:33:37 2002


Process memory map:
  2000000000030000-200000000003c000 rw-p 0000000000000000 00:00 0
        [2000000000030000-2000000000034000]           1 page  on node  21  MEMORY|DIRTY
        [2000000000034000-200000000003c000]           2 pages on node  12  MEMORY|DIRTY|SHARED

  2000000000044000-2000000000060000 rw-p 0000000000000000 00:00 0
        [2000000000044000-2000000000050000]           3 pages on node  12  MEMORY|DIRTY|SHARED
                                          ...
_______________________________________________________________________________
_______________________________________________________________________________
Exit:  ft.C.8
Pid: 2310       Fri Aug 30 14:33:37 2002


Process memory map:
  2000000000030000-200000000003c000 rw-p 0000000000000000 00:00 0
        [2000000000030000-2000000000034000]           1 page  on node  25  MEMORY|DIRTY
        [2000000000034000-200000000003c000]           2 pages on node  12  MEMORY|DIRTY|SHARED

  2000000000044000-2000000000060000 rw-p 0000000000000000 00:00 0
        [2000000000044000-2000000000050000]           3 pages on node  12  MEMORY|DIRTY|SHARED
        [2000000000050000-2000000000054000]           1 page  on node  25  MEMORY|DIRTY

                                            ...
_______________________________________________________________________________
_______________________________________________________________________________
Exit:  ft.C.8
Pid: 2307       Fri Aug 30 14:33:37 2002
```

```
Process memory map:
  2000000000030000-200000000003c000 rw-p 0000000000000000 00:00 0
        [2000000000030000-2000000000034000]            1 page  on node  30  MEMORY|DIRTY
        [2000000000034000-200000000003c000]            2 pages on node  12  MEMORY|DIRTY|SHARED

  2000000000044000-2000000000060000 rw-p 0000000000000000 00:00 0
        [2000000000044000-2000000000050000]            3 pages on node  12  MEMORY|DIRTY|SHARED
        [2000000000050000-2000000000054000]            1 page  on node  30  MEMORY|DIRTY
                                            ...
_______________________________________________________________________________
_______________________________________________________________________________
Exit:  ft.C.8
Pid: 2308       Fri Aug 30 14:33:37 2002


Process memory map:
  2000000000030000-200000000003c000 rw-p 0000000000000000 00:00 0
        [2000000000030000-2000000000034000]            1 page  on node   0  MEMORY|DIRTY
        [2000000000034000-200000000003c000]            2 pages on node  12  MEMORY|DIRTY|SHARED

  2000000000044000-2000000000060000 rw-p 0000000000000000 00:00 0
        [2000000000044000-2000000000050000]            3 pages on node  12  MEMORY|DIRTY|SHARED
        [2000000000050000-2000000000054000]            1 page  on node   0  MEMORY|DIRTY
                                           ...
```

For more information on the `dlook` command, see the `dlook` man page.

dplace

The `dplace` command allow you to control the placement of a process onto specified CPUs as follows:

`dplace [-c` *cpu_numbers*`] [-s` *skip_count*`] [-n` *process_name*`]`
`[-x` *skip_mask*`] [-p` *placement_file*`] command [`*command-args*`]`

`dplace -q`

Scheduling and memory placement policies for the process are set up according to `dplace` command line arguments.

By default, memory is allocated to a process on the node on which the process is executing. If a process moves from node to node while it running, a higher percentage of memory references are made to remote nodes. Because remote accesses typically have higher access times, process performance can be diminished.

You can use the `dplace` command to bind a related set of processes to specific CPUs or nodes to prevent process migrations. In some cases, this improves performance since a higher percentage of memory accesses are made to local nodes.

Processes always execute within a CpuMemSet. The CpuMemSet specifies the CPUs on which a process can execute. By default, processes usually execute in a CpuMemSet that contains all the CPUs in the system (for detailed information on CpusMemSets, see Chapter 4, "CPU Memory Sets and Scheduling", page 87).

The `dplace` command invokes a kernel hook (that is, a process aggregate or PAGG) to create a placement container consisting of all the CPUs (or a or a subset of CPUs) of the CpuMemSet. The `dplace` process is placed in this container and by default is bound to the first CPU of the CpuMemSet associated with the container. Then `dplace` invokes `exec` to execute the command.

The command executes within this placement container and remains bound to the first CPU of the container. As the command forks child processes, they inherit the container and are bound to the next available CPU of the container.

If you do not specify a placement file, `dplace` binds processes sequentially in a round-robin fashion to CPUs of the placement container. For example, if the current CpuMemSet consists of physical CPUs 2, 3, 8, and 9, the first process launched by `dplace` is bound to CPU 2. The first child process forked by this process is bound to CPU 3, the next process (regardless whether it is forked by parent or child) to 8, and so on. If more processes are forked than there are CPUs in the CpuMemSet, binding starts over with the first CPU in the CpuMemSet.

For more information on `dplace`(1) and examples of how to use the command, see the `dplace`(1) man page.

The `dplace`(1) command accepts the following options:

`-c` *cpu_numbers* The *cpu_numbers* variable specifies a list of CPU ranges, for example: "-c1", "-c2-4", "-c1, 4-8, 3". CPU numbers are **not** physical CPU numbers. They are logical CPU numbers that are relative to the CPUs that are in the set of allowed CPUs as specified by the current CpuMemSet or `runon`(1) command. CPU numbers start at 0. If this option is not specified, all CPUs of the

	current CpuMemSet are available. Note that a previous `runon` command may be used to restrict the available CPUs.
`-s` *skip_count*	Skips the first *skip_count* processes before starting to place processes onto CPUs. This option is useful if the first *skip_count* processes are “shepherd" processes that are used only for launching the application. If *skip_count* is not specified, a default value of 0 is used.
`-n` *process_name*	Only processes named *process_name* are placed. Other processes are ignored and are not explicitly bound to CPUs. **Note:** The *process_name* argument is the basename of the executable.
`-x` *skip_mask*	Provides the ability to skip placement of processes. The `skip_mask` argument is a bitmask. If bit N of `skip_mask` is set, then the N+1th process that is forked is not placed. For example, setting the mask to 6 causes the second and third processes from being placed. The first process (the process named by the `command`) will be assigned to the first CPU. The second amd third processes are not placed. The fourth process is assigned to the second CPU, and so on. This option is useful for certain classes of threaded applications that spawn a few helper processes that typically do not use much CPU time. **Note:** Intel OpenMP applications currently should be placed using the `-x` option with a `skip_mask` of *6* (`-x`*6*). This could change in future versions of OpenMP.
`-p` *placement_file*	Specifies a placement file that contains additional directives that are used to control process placement. (Not yet implemented).
`command` [*command-args*]	Specifies the command you want to place and its arguments.

`-q`	Lists the global count of the number of active processes that have been placed (by `dplace`) on each CPU in the current cpuset. Note that CPU numbers are logical CPU numbers within the cpuset, **not** physical CPU numbers.

Example 6-1 Using `dplace` command with MPI Programs

You can use the `dplace` command to improve placement of MPI programs on NUMA systems and verify placement of certain data structures of a long running MPI program by running a command such as the following:

`mpirun -np 64 /usr/bin/dplace -s1 -c 0-63 ./a.out`

You can then use the `dlook`(1) command to verify placement of certain data structures of long running MPI program by using the `dlook` command in another window on one of the slave thread PIDs to verify placement. For more information on using the `dlook` command, see "`dlook`", page 115 and the `dlook`(1) man page.

Example 6-2 Using `dplace` command with OpenMP Programs

To run an OpenMP program on logical CPUs 4 through 7 within the current CpuMemSet, perform the following:

```
efc -o prog -openmp -O3 program.f
setenv OMP_NUM_THREADS 4
dplace -x6 -c4-7 ./prog
```

The `dplace`(1) command has a static load balancing feature so that you do not necessarily have to supply a CPU list. To place `prog1` on logical CPUs 0 through 3 and `prog2` on logical CPUs 4 through 7, perform the following:

```
setenv OMP_NUM_THREADS 4
dplace -x6 ./prog1 &
dplace -x6 ./prog2 &
```

You can use the `dplace -q` command to display the static load information.

Example 6-3 Using `dplace` command with Linux Commands

The following examples assume that the command is executed from a shell running in a CpuMemSet consisting of physical CPUs 8 through 15.

Command	Run Location
`dplace -c2 date`	Runs the `date` command on physical CPU 10.

`dplace make linux`	Runs `gcc` and related processes on physical CPUs 8 through 15.
`dplace -c0-4,6 make linux`	Runs `gcc` and related processes on physical CPUs 8 through 12 or 14.
`runon 4-7 dplace app`	The `runon` command restricts execution to physical CPUs 12 through 15. The `dplace` sequentially binds processes to CPUs 12 through 15.

topology

The `topology`(1) command provides topology information about your system. Topology information is extracted from information in the`/dev/hw` directory. Unlike IRIX, in Linux the hardware topology information is implemented on a `devfs` filesystem rather than on a `hwgraph` filesystem. The `devfs` filesystem represents the collection of all significant hardware connected to a system, such as CPUs, memory nodes, routers, repeater routers, disk drives, disk partitions, serial ports, Ethernet ports, and so on. The `devfs` filesystem is maintained by system software and is mounted at `/hw` by the Linux kernel at system boot.

Applications programmers can use the `topology` command to help execution layout for their applications. For more information, see the `topology`(1) man page.

Output from the `topology` command is similar to the following: (Note that the following output has been abbreviated.)

```
% topology
Machine parrot.americas.sgi.com has:
64 cpu's
32 memory nodes
8 routers
8 repeaterrouters

The cpus are:
cpu 0 is /dev/hw/module/001c07/slab/0/node/cpubus/0/a
cpu 1 is /dev/hw/module/001c07/slab/0/node/cpubus/0/c
cpu 2 is /dev/hw/module/001c07/slab/1/node/cpubus/0/a
cpu 3 is /dev/hw/module/001c07/slab/1/node/cpubus/0/c
cpu 4 is /dev/hw/module/001c10/slab/0/node/cpubus/0/a
                         ...
The nodes are:
```

```
node 0 is /dev/hw/module/001c07/slab/0/node
node 1 is /dev/hw/module/001c07/slab/1/node
node 2 is /dev/hw/module/001c10/slab/0/node
node 3 is /dev/hw/module/001c10/slab/1/node
node 4 is /dev/hw/module/001c17/slab/0/node
                        ...
The routers are:
/dev/hw/module/002r15/slab/0/router
/dev/hw/module/002r17/slab/0/router
/dev/hw/module/002r19/slab/0/router
/dev/hw/module/002r21/slab/0/router
                        ...
The repeaterrouters are:
/dev/hw/module/001r13/slab/0/repeaterrouter
/dev/hw/module/001r15/slab/0/repeaterrouter
/dev/hw/module/001r29/slab/0/repeaterrouter
/dev/hw/module/001r31/slab/0/repeaterrouter
                       ...
The topology is defined by:
/dev/hw/module/001c07/slab/0/node/link/1 is /dev/hw/module/001c07/slab/1/node
/dev/hw/module/001c07/slab/0/node/link/2 is /dev/hw/module/001r13/slab/0/repeaterrouter
/dev/hw/module/001c07/slab/1/node/link/1 is /dev/hw/module/001c07/slab/0/node
/dev/hw/module/001c07/slab/1/node/link/2 is /dev/hw/module/001r13/slab/0/repeaterrouter
/dev/hw/module/001c10/slab/0/node/link/1 is /dev/hw/module/001c10/slab/1/node
/dev/hw/module/001c10/slab/0/node/link/2 is /dev/hw/module/001r13/slab/0/repeaterrouter
```

Installing NUMA Tools

To use the `dlook`(1), `dplace`(1), and `topology`(1) commands, you must load the `numatools` kernel module. Perform the following steps:

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

   ```
   chkconfig --add numatools
   ```

2. To turn on `numatools`, enter the following command:

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

This step will be done automatically for subsequent system reboots when `numatools` are configured on by using the `chkconfig`(8) utility.

The following steps are required to disable `numatools`:

1. To turn off `numatools`, enter the following:

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

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

   ```
   chkconfig --del numatools
   ```

Index