IRIX® Admin: Resource Administration

007–3700–009

CONTRIBUTORS

Written by Terry Schultz

Edited by Susan Wilkening

Illustrated by Chris Wengelski

Production by Glen Traefald

Engineering contributions by Tom Goozen, Sharif Islam, Marlys Kohnke, Tina Liang, Dennis Parker, Michael Sanford, Dan Stekloff, and Sam Watters

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

New Features in This Manual

This rewrite of *IRIX Admin: Resource Administration* supports the 6.5.14 release of IRIX.

New Features Documented

This release describes how to use the a `-q` *cpuset_name* `-p` option to the `cpuset`(1) command that allows you to see the properties of particular cpuset, such as, the number of processes and CPUs associated with the specified cpuset.

This release adds the `cpusetGetProperties()` and `cpusetFreeProperties()` functions to the Cpuset System library. The `cpusetGetProperties()` function allows you to retrieve various properties associated with a cpuset. The `cpusetFreeProperties()` functions allows you to release the memory used by a `cpuset_Properties_t` structure. For more information, see "Obtaining the Properties Associated with a Cpuset", page 64, and the `cpusetGetProperties`(3x) and `cpusetFreeProperties`(3x) man pages.

Major Documentation Changes

Changes to this document for the IRIX 6.5.14 release include a new section called "Writing a User Exit", page 113 and a minor correction to the `waitjob`(2) man page information in "`waitjob`", page 136.

Record of Revision

Version	Description
001	July 1999 Draft version.
002	January 2000 Supports the IRIX 6.5.7 release.
003	April 2000 Supports the IRIX 6.5.8 release.
004	August 2000 Supports the IRIX 6.5.9 release.
005	November 2000 Supports the IRIX 6.5.10 release.
006	February 2001 Supports the IRIX 6.5.11 release.
007	May 2001 Supports the IRIX 6.5.12 release.
008	August 2001 Supports the IRIX 6.5.13 release.
009	November 2001 Supports the IRIX 6.5.14 release.

Contents

Figures

Tables

Examples

About This Manual

This publication documents the IRIX 6.5.14 operating system running on SGI server systems.

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

This manual contains the following chapters:

- Chapter 1, "Process Limits", page 1
- Chapter 2, "Job Limits", page 5
- Chapter 3, "Miser Batch Processing System", page 29
- Chapter 4, "Cpuset System", page 49
- Chapter 5, "Comprehensive System Accounting", page 71
- Chapter 6, "IRIX Memory Usage", page 129
- Appendix A, "Programming Guide for Resource Management", page 133

Related Publications

This guide is part of the *IRIX Admin* manual set, which is intended for administrators: those who are responsible for servers, multiple systems, and file structures outside the user's home directory and immediate working directories. If you maintain systems for others or if you require more information about IRIX than is in the end-user manuals, these guides are for you. The *IRIX Admin* guides are available through the IRIS InSight online viewing system. The set consists of these volumes:

- *IRIX Admin: Software Installation and Licensing* - Explains how to install and license software that runs under IRIX, the SGI implementation of the UNIX operating system. Contains instructions for performing miniroot and live installations using `inst`(1M), the command line interface to the IRIX installation utility. Identifies the licensing products that control access to restricted applications running under IRIX and refers readers to licensing product documentation.

- *IRIX Admin: System Configuration and Operation* - Lists good general system administration practices and describes system administration tasks, including configuring the operating system; managing user accounts, user processes, and disk resources; interacting with the system while in the PROM monitor; and tuning system performance.
- *IRIX Admin: Disks and Filesystems* - Explains disk, filesystem, and logical volume concepts. Provides system administration procedures for SCSI disks, XFS and Extent File System (EFS) filesystems, XLV logical volumes, and guaranteed-rate I/O.
- *IRIX Admin Networking and Mail* - Describes how to plan, set up, use, and maintain the networking and mail systems, including discussions of sendmail, UUCP, SLIP, and PPP.
- *IRIX Admin: Backup, Security and Accounting* - Describes how to back up and restore files, how to protect your system's and network's security, and how to track system usage on a per-user basis.
- *IRIX Admin: Resource Administration* - Provides an introduction to system resource administration and describes how to use and administer various IRIX resource management features, such as IRIX job limits, the Miser Batch Processing System, the Cpuset System, and Comprehensive System Accounting (CSA).
- *IRIX Admin: Peripheral Devices* - Describes how to set up and maintain the software for peripheral devices such as terminals, modems, printers, and CD-ROM and tape drives.
- *IRIX Admin: Selected Reference Pages* – (not available in InSight) – Provides concise man page information on the use of commands that may be needed while the system is down. Generally, each man page covers one command, although some man pages cover closely related commands. Man pages are available online through the `man`(1) command.

Obtaining Publications

To obtain SGI documentation, go to the SGI Technical Publications Library at:

`http://techpubs.sgi.com`

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, please tell us. 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 us 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 World Wide Web page:

  `http://techpubs.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 Pkwy., M/S 535
  Mountain View, California 94043–1351

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

We value your comments and will respond to them promptly.

Chapter 1

Process Limits

Standard system resource limits are applied so that each login process receives the same process-based limits at the time the process is created. This chapter describes process limits and contains the following sections:

- "Process Limits Overview", page 1
- "Using `csh` and `sh` to Limit Resource Consumption", page 1
- "Using systune to Display and Set Process Limits", page 2
- "Additional Process Limits Parameters", page 4

Process Limits Overview

The IRIX operating system supports limits on individual processes. Limits on the consumption of a variety of system resources by a process and each process it creates may be obtained with the `getrlimit`(2) system call and set with the `setrlimit`(2) system call.

Each call to either `getrlimit` or `setrlimit` identifies a specific resource to be operated upon as well as a resource limit. A resource limit is a pair of values: one specifying the current (soft) limit, the other a maximum (hard) limit. Soft limits may be changed by a process to any value that is less than or equal to the hard limit. A process may (irreversibly) lower its hard limit to any value that is greater than or equal to the soft limit.

Using csh and sh to Limit Resource Consumption

The `csh` or `sh` `limit -h` *resource max-use* commands can be used to limit the resource consumption by the current process or any process it spawns.

These commands limit the consumption by the current process and each process it creates to not individually exceed maximum-use on the specified resource. If no maximum-use is given, then the current limit is printed; if no resource is given, then all limitations are given. If the `-h` flag is given, the hard (maximum) limits are used instead of the current limits. The hard limits impose a ceiling on the values of the current limits. To raise maximum (hard) limits, you must have the CAP_PROC_MGT capability.

For additional information, see the csh(1) and sh(1) man pages. For more information on the capability mechanism that provides fine grained control over the privileges of a process, see the capability(4) and capabilities(4) man pages.

Using systune to Display and Set Process Limits

Table 1-1 shows the process limits supported by the IRIX operating system.

Table 1-1 Process Limits

Limit Name	Symbolic ID	Units	Description	Enforcement
rlimit_cpu_cur rlimit_cpu_max	RLIMIT_CPU	seconds	Maximum number of CPU seconds the process is allowed	Process termination via SIGXCPU signal
rlimit_fsize_cur rlimit_fsize_max	RLIMIT_FSIZE	bytes	Maximum size of file that can be created by process	Write/expansion attempt fails with errno set to EFBIG
rlimit_data_cur rlimit_data_max	RLIMIT_DATA	bytes	Maximum process heap size	brk(2) calls fail with errno set to ENOMEM
rlimit_stack_cur rlimit_stack_max	RLIMIT_STACK	bytes	Maximum process stack size	Process termination via SIGSEGV signal
rlimit_ core_cur rlimit_core_max	RLIMIT_CORE	bytes	Maximum size of a core file that can be created by process	Writing of core file terminated at limit
rlimit_nofile_cur rlimit_nofile_max	RLIMIT_NOFILE	file descriptors	Maximum number of open file descriptors process can have	open(2) attempts file with errno set to EMFILE

Limit Name	Symbolic ID	Units	Description	Enforcement
`rlimit_vmem_cur` `rlimit_vmem_max`	`RLIMIT_VMEM`	bytes	Maximum process address space	`brk`(2) and `mmap`(2) calls fail with `errno` set to `ENOMEM`
`rlimit_rss_cur` `rlimit_rss_max`	`RLIMIT_RSS`	bytes	Maximum size of resident set size of the process	Resident pages above limit become prime swap candidates
`rlimit_pthread_cur` `rlimit_pthread_max`	`RLIMIT_PTHREAD`	threads	Maximum number of threads that process can create	Thread creation fails with `errno` set to `EAGAIN`

You can use the `systune` *resource* command to view and set systemwide default values for process limits. The *resource* group contains the following variables:

```
rlimit_cpu_cur
rlimit_cpu_max
rlimit_fsize_cur
rlimit_fsize_max
rlimit_data_cur
rlimit_data_max
rlimit_stack_cur
rlimit_stack_max
rlimit_core_cur
rlimit_core_max
rlimit_nofile_cur
rlimit_nofile_max
rlimit_vmem_cur
rlimit_vmem_max
rlimit_rss_cur
rlimit_rss_max
rlimit_pthread_cur
rlimit_pthread_max
```

For additional information, see the `systune`(1M) man page.

If job limits software is installed and running on the system, you can choose to set user-based process limits values in the user limits database (ULDB). Both current and maximum values, such as `rlimit_cpu_cur` and `rlimit_cpu_max` can be specified. Values in the ULDB override the system defaults set by the `systune`(1M) command.

For additional information on the ULDB, see "User Limits Database", page 11.

Additional Process Limits Parameters

IRIX has configurable parameters for certain system limits. For example, you can set maximum values for each process (its core or file size), the number of groups per user, the number of resident pages, and so forth. The `maxup` and `cpulimit_gracetime` are described below. All parameters are set and defined in `/var/sysgen/mtune`.

`maxup`	Maximum number of processes per user
`cpulimit_gracetime`	Process and job limit grace period

For additional information on the `maxup` parameter and other “System Limits Parameters”, see *IRIX Admin: System Configuration and Operation*.

The `cpulimit_gracetime` parameter establishes a grace period for processes that exceed the CPU time limit. You should set it to the number of seconds that a process will be allowed to run after exceeding the limit. When `cputlimit_gracetime` is not set (that is, when it is zero), any process that exceeds either the process or job CPU limit will be sent a `SIGXCPU` signal. The kernel will periodically send a `SIGXCPU` signal to that process as long as it continues to execute. Since a process can register to handle a `SIGXCPU` signal, the process can effectively ignore the CPU limit.

If you use the `systune`(1M) command to set the `cpulimit_gracetime` parameter to a nonzero value, its behavior changes. When a process exceeds the CPU limit, the kernel sends a `SIGXCPU` signal to a process only once. The process can register for this signal and then perform any cleanup and shutdown operations it wants to perform. If the process is still running after accruing `cpulimit_gracetime` more seconds of CPU time, the kernel terminates the process with a `SIGKILL` signal.

Chapter 2

Job Limits

Standard system resource limits are set up so that each process receives the same process-based limits at the time the process is created. While limits on individual processes are useful, they do not restrict individual users to a given share of the system. With the IRIX kernel job limits feature, all processes associated with a particular login session or batch submission are encapsulated as a single logical unit called a job. The job is the container used to group processes by login session. Limits on resource usage are applied on a per user basis for a particular job and these limits are enforced by the kernel. All processes are associated with a particular job and are identified by a unique job identifier (job ID). The processes belonging to a particular job can be limited, controlled, queried, and accounted for as a unit. This allows a system administrator to set job-specific limits on CPU time, memory, file space, and other system resources. The user limits database (ULDB) allows user-specific limits for jobs. If no ULDB is defined, job limits are the same for all jobs. Job limits software can help maximize utilization of larger systems in a multiuser environment.

This chapter contains the following sections:

- "Read Me First", page 5
- "Job Limits Overview", page 6
- "Job Limits Supported", page 9
- "User Limits Database", page 11
- "Running Job Limits with Message Passing Interface (MPI) Jobs", page 25
- "Installing Job Limits", page 26
- "Job Limits Man Pages", page 26
- "Error Messages", page 28

Read Me First

The sections in this chapter contain information about installing job limits software on your system. You should reference them in the order they are listed here:

1. For a general description of jobs and job limits, see "Job Limits Overview", page 6, and "Job Limits Supported", page 9.
2. To install the job limits package, see "Installing Job Limits", page 26.
3. For information about writing a user limits directives input file *infile* and creating the user limits database (ULDB), see "Creating the User Limits Directives Input File", page 13, and "Creating the User Limits Database", page 12, respectively.

   For a list of man pages related to job limits, see "Job Limits Man Pages", page 26.
4. For information on how to use the `systune` *joblimits* command to set systemwide default values for job limits, see "Using `systune` to Display and Set Job Limits", page 17.
5. For information on how to view job limits on a system, see "User Commands for Viewing and Setting Job Limits", page 18.
6. For information on troubleshooting your job limits installation, see "Troubleshooting Job Limits", page 26.
7. For information on application programming interfaces, see "Application Programming Interface for Job Limits", page 133, and "Application Programming Interface for the ULDB", page 136.

Job Limits Overview

Job limits software helps ensure that each user has access to the appropriate amount of system resources such as CPU time and memory and makes sure that users do not exceed their allotted amount. Job limits software can improve system throughput and utilization by restricting how much of a machine each user can use. For information on user-based job limits supported in IRIX, see "Job Limits Supported", page 9.

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 create an original shell process and multiple processes flow from that original point of entry. The kernel job provides a means to limit 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 2-1, page 7, shows the point of entry processes that initiate the creation of jobs.

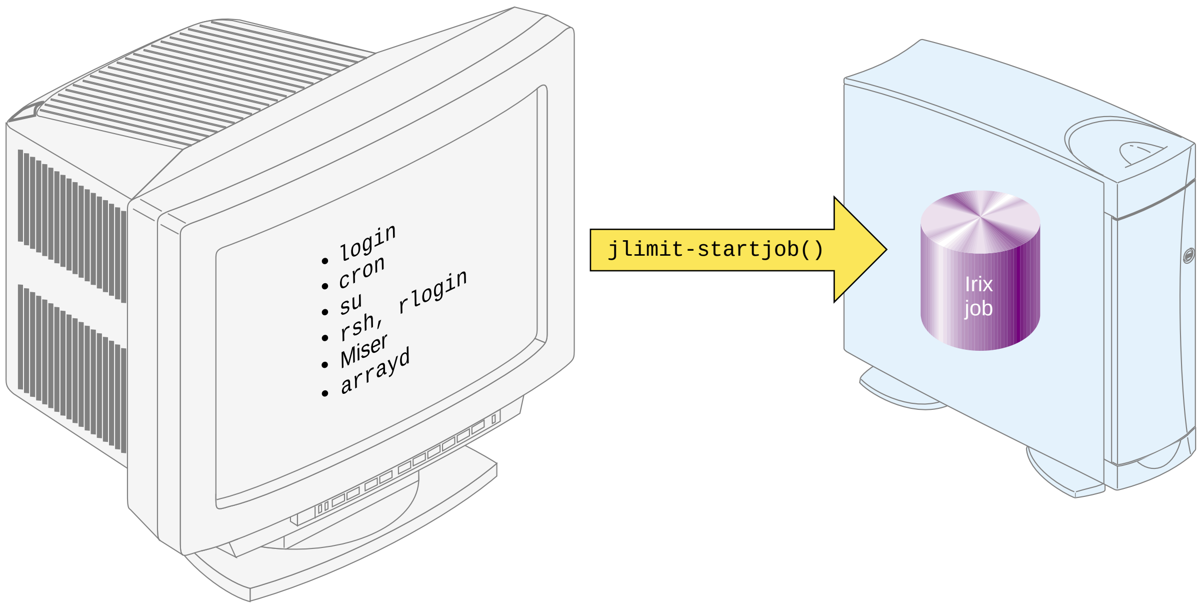

Figure 2-1 Point of Entry Processes

IRIX job limits have 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 and limits from its parent process.
- All point of entry processes (job initiators) create a new job and set the job limits appropriately.
- Users can raise and lower their own job limits within maximum values specified by the system administrator.
- The job initiator performs authentication and security checks.

The process control initialization process (`init`(1M)) and startup scripts called by `init` are not part of a job and have a job ID of zero.

Note: The upper bits of the job ID are used to indicate the machine ID. The job ID contains the array services machine ID (`asmchid`). Array services are started by the init process and large job IDs are created. To the administrator, this may seem like large job ID values appear without explanation because they have not set the machine ID. For more information on the `asmchid` parameter, see Appendix A, "IRIX Kernel Tunable Parameters", in the *IRIX Admin: System Configuration and Operation* and the `arsctl`(2) and `newarraysess`(2) man pages.

Note: The existing IRIX commands `jobs`(1), `fg`(1), and `bg`(1) man pages apply to shell "jobs" and are not related to IRIX kernel job limits.

Note: Job initiators like secure shell that are not developed by SGI might not initiate an IRIX kernel job.

Figure 2-2 shows two limit domains. Limit domains are a way to categorize work. The job initiators shown in Figure 2-1, page 7, can be categorized as either interactive or batch processes. Limit domain names are defined by the system administrator when the user limits database (ULDB) is created. Applications that use the ULDB to retrieve job limits information expect to find limit information with specific names. These names are defined by convention. For additional information on limit domains and the ULDB, see "User Limits Database", page 11.

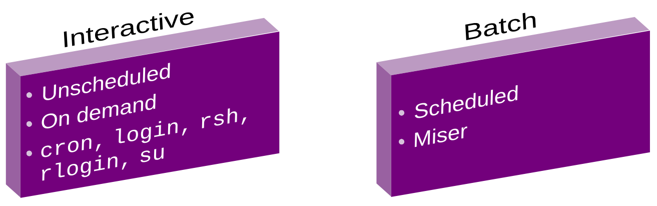

Figure 2-2 Limit Domains

The IRIX operating system provides a number of commands that provide information about the memory usage on a system. The job limits `jstat`(1) command reports the current usage and highwater memory values of all concurrently running processes within a job. For more information on memory usage in IRIX, see Chapter 6, "IRIX Memory Usage", page 129. For more information on the `jstat`(1) command, see "`jstat`", page 22.

Job Limits Supported

Table 2-1 shows job limits supported by the IRIX operating system. Each limit restricts the use of a particular system resource for all the processes contained within a job. Job limits software also introduces a limit unique to jobs called `JLIMIT_NUMPROC` that controls the number of processes in a job.

Table 2-1 Job Limits

Limit Name	Symbolic ID	Units	Description	Enforcement
`jlimit_nproc_cur` `jlimit_nproc_max`	`JLIMIT_NUMPROC`	processes	Maximum number of processes within the job	Process creation by any job fails with `errno` set to `EAGAIN`
`jlimit_nofile_cur` `jlimit_nofile_max`	`JLIMIT_NOFILE`	file descriptors	Maximum total number of open file descriptors all processes in job can have	`open`(2) calls by any job fail with `errno` set to `EMFILE`
`jlimit_rss_cur` `jlimit_rss_max`	`JLIMIT_RSS`	bytes	Maximum total resident set size for all processes in a job	Resident pages above limit become prime swap candidates
`jlimit_vmem_cur` `jlimit_vmem_max`	`JLIMIT_VMEM`	bytes	Maximum total address space for all processes in a job	The `brk`(2) and `mmap`(2) calls in any job fail with `errno` set to `ENOMEM`

Limit Name	Symbolic ID	Units	Description	Enforcement
jlimit_data_cur jlimit_data_max	JLIMIT_DATA	bytes	Maximum total heap size for all processes in job	The brk(2) calls in any job fail with errno set to ENOMEM
jlimit_cpu_cur jlimit_cpu_max	JLIMIT_CPU	seconds	Maximum number of CPU seconds allowed for all processes in a job	Termination of all processes in job via SIGXCPU signal. You can also use the cpulimit_gracetime parameter to alter signalling behavior, see "cpulimit_gracetime", page 11.
jlimit_pmem_cur jlimit_pmem_max	JLIMIT_PMEM	bytes	Maximum total resident set size for all processes in a job	Termination of all processes in job via SIGKILL signal

getjlimit and setjlimit

Limits on the consumption of system resources by a job, shown in Table 2-1, page 9, may be obtained with the getjlimit(2) function and set by the setjlimit(2) function. The getjlimit function gets the current and maximum job limits values for the specified job. The CAP_MAC_READ capability is needed to retrieve values from jobs belonging to other users.

The setjlimit(2) function sets the current and maximum job limits values for the specified job. If the current job is different from the job being requested, the setjlimit function checks for the CAP_MAC_WRITE capability. If the maximum (hard) limits are being raised, the setjlimit function checks for the CAP_PROC_MGT capability.

For additional information, see the getjlimit(2) man page. For more information on the capability mechanism that provides fine grained control over the privileges of a process, see the capability(4) and capabilities(4) man pages.

waitjob

The `waitjob` mechanism allows a batch processing system to find out job limit information for jobs that exit abnormally. The `waitjob` function obtains information about a terminated job that has been set with `setwaitjobpid` argument to wait. For more information on the `waitjob`(2) and `setwaitjobpid`(2) calls, see "Application Programming Interface for Job Limits", page 133 and "Application Programming Interface for the ULDB", page 136, respectively, and the `waitjob`(2) and `setwaitjobpid`(2) man pages.

systune

You can use the `systune` *joblimits* command to set system-wide defaults. For additional information, see "Using `systune` to Display and Set Job Limits", page 17 and the `systune`(1M) man page.

cpulimit_gracetime

The `cpulimit_gracetime` parameter establishes a grace period for processes that exceed the CPU time limit. The `cpulimit_gracetime` parameter controls the signalling behavior associated with the CPU limit. For additional information on the `cpulimit_gracetime` parameter, see "Additional Process Limits Parameters", page 4.

Job limits software works in a manner similar to process limits when dealing with the `cpulimit_gracetime`. As a process executes, the CPU usage increases. When the limit is reached, the `SIGXCPU` signal is sent individually to each process when it executes. When the `SIGXCPU` is sent to a process, the grace period goes into effect for that process. If the process is still executing when the grace period expires, it is terminated with the `SIGKILL` signal. Only the processes in a job that are executing, are sent a `SIGXCPU` signal. Each process in a job gets an individual grace period. Therefore, the `SIGXCPU` signal is not sent en masse to all processes in a job.

User Limits Database

The User Limits Database (ULDB) contains job limits information which allows a system administrator to control access to a machine on a per user basis. Job initiators, the applications that initiate new jobs on the system like `login`, `rsh`, `rlogin`, `cron`, and workload management systems like Miser, retrieve job limits values from the ULDB for a particular user and use the information to set limits, appropriately.

For more information on job initiators, see "Job Limits Overview", page 6.

The ULDB is used to set job limit and process limit values for jobs when the job limits package is installed. If job limits are not installed, process limits are handled by the current resource limits functionality.

Domain defaults apply to all users unless there is a "user" entry that describes values for that user. User specific values override the domain defaults. Values in the ULDB override the system default values for both job limits and process limits.

This section describes the commands used to create, maintain, and display the contents of the ULDB and the library application programming interface (API), which allows applications access to the ULDB information.

Note: The ULDB configuration file contained in the `/etc/jlimits.in` file contains a template you can follow when setting up the ULDB on your system.

The `/etc` directory also contains the `jlimits` and `jlimits.m` files. The `jlimits.in` file is parsed into the colon delimited `jlimits` file, which is used to load job limits into the local ULDB `jlimits.m` file or into the NIS master map. The `jlimits` file is automatically generated by the `genlimits`(1M) command. The `jlimits.m` file is the local ULDB mdbm file.

Creating the User Limits Database

The command to create the ULDB is as follows:

`genlimits [-i` *infile*`] [-l] [-m] [-L` *local_database*`] [-N` *nisfile*`] [-v]`

The `genlimits` command parses the formatted ASCII user limits directives input file (*infile*) into a colon-delimited ASCII file, which can be used to create one of the following output formats:

- Network Information Service (NIS) master server map (`-m` option)
- Local database for NIS or direct (non-NIS) use (`-l` option)

The `genlimits` command accepts the following options:

`-i` *infile*	Identifies the location of the user limits directives input file. If you do not specify the `-i` option, the default file is `/etc/jlimits.in`.
`-l`	Creates a local database for Network Information Service (NIS) or direct (non-NIS) use. When NIS is enabled, the local database contains local entries which override or supplement entries from the NIS server. When NIS is not enabled, the local database contains information to set limits on the system. By default, this database is in the `/etc/jlimits.m` file. You cannot use the `-l` option with the `-m` option.
`-m`	Creates the NIS master server map. It generates and stores the map in the standard NIS map location. You cannot override this location. You cannot use the `-m` option with the `-l` option.
`-L` *local_database*	Specifies an alternate location for the local database. The `-L` option works in conjunction with the `-l` option.
`-N` *nisfile*	Specifies a different location for the created NIS database source input file. The default location is the `/etc/jlimits` file. You can use the `-N` *nisfile* option to create a new database without overwriting the existing `/etc/jlimits` file.
`-v`	Specifies verbose mode, which prints out messages describing actions of the `genlimits` command.

For additional information, see the `genlimits`(1M) man page.

Creating the User Limits Directives Input File

The user limits directive file contains the input to the `genlimits`(1M) command, defining the information on domains, limits, and users that will be used to generate the ULDB. This section describes how to write a user limits directives input file.

Comments

Any text following the # character is treated as a comment.

Numeric Limit Values

Numeric values can have a letter appended that indicate a multiplier that is applied to the numeric value provided to determine the limit value as follows:

Letter	**Multiplier Value**
k (kilo)	1024 (2**10)
m (mega)	1,048,576 (2**20)
g (giga)	1,073,741,824 (2**30)
t (tera)	1,099,511,627,776 (2**40)
H (hours)	3600
M (minutes)	60

- Use the k, m, g, and t multipliers when defining memory limits or other large values.
- Use the H and M multipliers when defining time values.

Multiplier values are defined in the `/usr/include/uldb.h` system include file.

There are no requirements that multipliers be use in the above manner.

Numeric limit values can also be specified as “unlimited” which indicates there is no upper limit for this particular limit type.

For additional information about creating the ULDB, see the `genlimits`(1M) man page.

Domain Directives

Each limit domain that is referenced in the ULDB must first be identified using the "domain" directive. The directive provides the ASCII domain name and a list of the default limit values for the domain. An example domain directive follows:

```
domain domain_name {
     limit_name = value
     limit_name:machname = value
     ...
  }
```

Certain domain names are reserved for user job limits. Other domain names may be created and used for special purposes. The following list contains reserved domain names:

Reserved Domain Name	Description
`interactive`	Used by interactive job initiators such as `telnet` and `login`
`batch`	A generic batch domain used as secondary choice for all workload management software
`miser`	The domain used when submitting work to Miser
`nqe`	The domain used when submitting work to NQE
`lsf`	The domain used when submitting work to LSF

User Directives

The "user" directive specifies a set of limits for an individual user. The user name must identify a valid login account. The `uid` value is optional. If `uid` is specified, the `genlimits` command verifies that the `uid` provided matches the `uid` defined for the user on the machine where `genlimits` executes. Domain clauses identify each domain for which the user will have unique limit values. The domain listed in the user directive must already be defined in a prior domain directive. The syntax and semantics of the domain clause is the same as the domain directive. It is not necessary to provide user directives for all users on the system. If there is no user directive for a queried user or there are no values for a queried domain, the default values for that domain are returned. An example user directive follows:

```
user user_name[:uid] {
      domain_name {
         limit_name = value
         limit_name:machname = value
         ...
      }
      domain_name {
         ...
      }
      ...
   }
```

The limit specifications for both the domain and user directives may include an optional machine name. Limit values specified with a machine name apply only to that machine. Limits without a machine name apply to all machines in the cluster. The directives input file can contain several occurrences of the same limit, each with a different name, as well as an occurrence without a machine name specified.

The `genlimits` command processes limit values with associated machine names differently depending on the type of database (see "Creating the User Limits Database", page 12) being generated:

- If the `-m` option is used to generate a NIS master map, limit values with associated machine names are ignored. Only clusterwide values without machine names are included in the database.
- If the `-l` option is used to generate a local database, the `genlimits` command selects the limit value with the name of the local machine if present. If there is no limit value with the local machine name, the `genlimits` command selects the clusterwide value with no machine name. You can determine the local machine name by running the `uname -n` command. For additional information on the `uname` command, see the `uname`(1) man page.

Setting Up a User Limits Directive Input File Example

Because the ULDB is completely rebuilt whenever the `genlimits` command is invoked, the input directive file must contain a complete representation of the database. When changes are needed, the system administrator must edit the user limits directives input file and then rebuild the database. Because domain defaults are used if there is no user entry for a particular user, the administrator only needs to provide user entries for named users to overwrite default values. The following example shows a user limits directives input file that specifies three limit types, two domains, and one user with individual limits. The ULDB only stores the limit values. The meaning of a value and the units it expresses are up to the application that uses the limit.

Note: If you are updating entries in the ULDB and they do not change the job limit values on your system, make sure that limit names used in the ULDB and limit names used in the `systune` *joblimits* group are exactly the same. For additional information, see "Troubleshooting Job Limits", page 26.

```
domain interactive {             # domain for interactive logins
   jlimit_cpu_cur = 60
   jlimit_cpu_max = 120          # limit interactive jobs to 120 CPU seconds
   jlimit_vmem_cur = 2m
   jlimit_vmem_max = 4m           # limit interactive jobs to 4 megabytes of virtual memory
   jlimit_numproc_cur =10
   jlimit_numproc_max = 20        # limit interactive jobs to 20 concurrent processes
}
domain batch {                    # domain for batch submissions
   jlimit_cpu_cur = 3600
   jlimit_cpu_max = 7200          # limit batch jobs to two hours of CPU time
   jlimit_vmem_cur = 128m
   jlimit_vmem_max = 256m         # limit batch jobs to 256 megabytes of memory
   jlimit_numproc_cur = unlimited
   jlimit_numproc_max = unlimited # no limit on processes in a batch job
}

user fred:123 {                   # User "fred" gets his own interactive CPU limits
   interactive  {                 #
      jlimit_cpu_cur = 300
      jlimit_cpu_max = 600        # "fred" needs to run longer jobs in interactive mode
    }
}
```

Using `systune` to Display and Set Job Limits

You can use the `systune` *joblimits* command to view and set systemwide default values for user job limits. The ULDB will override these values if it exists. The *joblimits* group contains the following variables:

```
jlimit_cpu_cur
jlimit_cpu_max
jlimit_data_cur
jlimit_data_max
jlimit_vmem_cur
jlimit_vmem_max
jlimit_rss_cur
jlimit_rss_max
jlimit_nofile_cur
jlimit_nofile_max
jlimit_numproc_cur
jlimit_numproc_max
jlimit_pmem_cur
jlimit_pmem_max
```

Output from the `systune` *joblimits* command follows:

```
$ systune joblimits
group: joblimits (statically changeable)
        jlimit_numproc_max = 1024 (0x400) ll
        jlimit_numproc_cur = 1024 (0x400) ll
        jlimit_nofile_max = 5000 (0x1388) ll
        jlimit_nofile_cur = 400 (0x190) ll
        jlimit_rss_max = 9223372036854775807 (0x7fffffffffffffff) ll
        jlimit_rss_cur = 9223372036854775807 (0x7fffffffffffffff) ll
        jlimit_vmem_max = 9223372036854775807 (0x7fffffffffffffff) ll
        jlimit_vmem_cur = 9223372036854775807 (0x7fffffffffffffff) ll
        jlimit_data_max = 9223372036854775807 (0x7fffffffffffffff) ll
        jlimit_data_cur = 9223372036854775807 (0x7fffffffffffffff) ll
        jlimit_cpu_max = 9223372036854775807 (0x7fffffffffffffff) ll
        jlimit_cpu_cur = 9223372036854775807 (0x7fffffffffffffff) ll
        jlimit_pmem_max = 9223372036854775807 (0x7fffffffffffffff) ll
        jlimit_pmem_cur = 9223372036854775807 (0x7fffffffffffffff) ll
```

The display information is described below:

- `jlimit_numproc` - Number of processes limit
- `jlimit_nofile` - Number of files limit
- `jlimit_rss` - Resident set size, default is in bytes
- `jlimit_vmem` - Virtual memory limit, default is in bytes
- `jlimit_data` - Data size, default is in bytes
- `jlimit_cpu` - CPU time, default in seconds.
- `jlimit_pmem` - Maximum resident set size for all processes in a job, default in bytes

For additional information, see the `systune`(1M) and `jlimit`(1) man pages.

User Commands for Viewing and Setting Job Limits

This section describes the following user commands which can be used to view and set job limits:

- "`showlimits`", page 19
- "`jlimit`", page 21
- "`jstat`", page 22

showlimits

The command to view limit information from the ULDB is as follows:

showlimits [-D] [-d] [-u *user_name*] [*domain_name*]

The `showlimits` command displays limits information from the user limits database (ULDB).

The `showlimits` command accepts the following options:

`-D`	Displays the names of all the domains defined in the ULDB. When you specify the `-D` option, the domain name and other options are ignored.
`-d`	Displays the domain default limits. When no options are specified, the `showlimits` command displays the default limits for all domains.
`-u` *user_name*	Displays the limits values for the specified user rather than the current user.
domain_name	Displays the limits values for the specified domain rather than all domains.

If no options are specified, the `showlimits` command displays the current limits information for the current user for all domains as shown below:

```
% showlimits

Domain interactive:
        jlimit_cpu_cur: unlimited
        jlimit_cpu_max: unlimited
        jlimit_data_cur: unlimited
        jlimit_data_max: unlimited
        jlimit_nofile_cur: 400
        jlimit_nofile_max: unlimited
        jlimit_vmem_cur: unlimited
        jlimit_vmem_max: unlimited
```

```
                jlimit_rss_cur: unlimited
                jlimit_rss_max: unlimited
                jlimit_pthread_cur: 2k
                jlimit_pthread_max: 65535
                jlimit_numproc_cur: 1k
                jlimit_numproc_max: 65535
                rlimit_cpu_cur: unlimited
                rlimit_cpu_max: unlimited
                rlimit_fsize_cur: unlimited
                rlimit_fsize_max: unlimited
                rlimit_data_max: unlimited
                rlimit_stack_cur: 64m
                rlimit_stack_max: unlimited
                rlimit_core_cur: unlimited
                rlimit_core_max: unlimited
                rlimit_nofile_cur: 200
                rlimit_nofile_max: unlimited
                rlimit_vmem_max: unlimited
                rlimit_rss_max: unlimited

        Domain batch:
                jlimit_cpu_cur: unlimited
                jlimit_cpu_max: unlimited
                jlimit_data_cur: unlimited
                jlimit_data_max: unlimited
                jlimit_nofile_cur: 400
                jlimit_nofile_max: unlimited
                jlimit_vmem_cur: unlimited
                jlimit_vmem_max: unlimited
                jlimit_rss_cur: unlimited
                jlimit_rss_max: unlimited
                jlimit_pthread_cur: 2k
                jlimit_pthread_max: 65535
                jlimit_numproc_cur: 1k
                jlimit_numproc_max: 65535
                rlimit_cpu_cur: unlimited
                rlimit_cpu_max: unlimited
                rlimit_fsize_cur: unlimited
                rlimit_fsize_max: unlimited
                rlimit_data_max: unlimited
                rlimit_stack_cur: 64m
```

```
rlimit_stack_max: unlimited
rlimit_core_cur: unlimited
rlimit_core_max: unlimited
rlimit_nofile_cur: 200
rlimit_nofile_max: unlimited
rlimit_vmem_max: unlimited
rlimit_rss_max: unlimited
```

Note: If the ULDB has changed after the user logged in, the current limits will not be effective. Current limits will be effective for any new users that login.

For a description of the job limit values, see Table 2-1, page 9. For a description of the process limit values, see Table 1-1, page 2.

For additional information, see the `showlimits`(1) man page.

`jlimit`

The command to display and set job limits is as follows:

`jlimit [-j` *job_id*`] [-h] [`*limit_name* `[`*value*`]]`

The `jlimit` command displays and changes limits on job resource usage. The current and maximum (hard) limits are set when a job starts from values that are contained in the user limits database (ULDB) information for the user. You can raise and lower your current limits within the range not to exceed your maximum limit. You can irrevocably lower your maximum limit. You must have the `CAP_PROC_MGT` capability to raise your maximum limit. Limit enforcement always occurs at the current limit regardless of your maximum limit value. See the `capability`(4) and `capabilities`(4) man pages for additional information on the capability mechanism that provides fine grained control over the privileges of a process.

The `jlimit` command accepts the following options:

`-j` *job_id*	Specifies a particular job ID for a job where limits are going to be changed. You must have the `CAP_MAC_WRITE` and `CAP_PROC_MGT` capabilities to change job limits for jobs that belong to other users. The job ID is printed out in hexadecimal. When the job ID is specified, the "`0x`" prefix is optional.

`-h`	Specifies that the maximum (hard) limit values for a job are displayed or modified. If you do not specify the `-h` option, the `jlimit` command displays or modifies current limit values.
limit_name [*value*]	Displays or sets the value for the specified limit: • If no limit name is specified, `jlimit` displays the values for all limits. • If the limit name is specified without a value, `jlimit` displays the value for the limit. • If both a limit name and a value are specified, `jlimit` sets the appropriate value for the limit.

If the `-j` option with a *job_id* argument is specified, the `jlimit` command prints out the following information:

```
 % jlimit -j 0x14
cputime: unlimited
datasize: unlimited
files: unlimited
vmemory: unlimited
ressetsize: unlimited
processes: 65535
```

For an explanation of the limit values, see Table 2-1, page 9.

For additional information, see the `jlimit`(1) man page.

jstat

The command to display job status information for active jobs is as follows:

```
jstat [-a] [-l] [-p]
jstat [-j job_id] [-l] [-p]
```

The `jstat` command accepts the following options:

`-a`	Displays information about all jobs.
`-j` *job_id*	Displays information only for the specified job ID (*job_id*).

`-l`	Displays limit information about the current or specified job including the current usage, current limit, and maximum limit.
`-p`	Displays information about each process that belongs to the current or specified job including the process ID, state, and executing command.
`-P`	Displays the memory limits information in pages rather than in bytes. This option is used with the `-l` option.

If neither the `-a` or `-j` *job_id* are used, the `jstat` command displays information on the current job.

If the `-l` option is specified, the `jstat` command prints out the current usage, high usage, current limit, and maximum limit information for the current job as shown below:

```
% jstat -l

JID             OWNER          COMMAND
--------------- -------------- --------------
0x5eac0000001bd terry            -csh

LIMIT NAME      USAGE          HIGH USAGE     CURRENT LIMIT  MAX LIMIT
--------------- -------------- -------------- -------------- --------------
cputime         1:05           1:05           unlimited      unlimited
datasize        400k           400k           unlimited      unlimited
files           10             35             400            5000
vmemory         44             201            unlimited      unlimited
ressetsize      340            357            unlimited      unlimited
processes       2              4              1024           1024
```

If the `-l` and `-P` options are specified, the `jstat` command will print out the same information that the `-l` option displays with the exception that memory values are shown in pages. SGI systems support multiple page sizes. For more information on pages sizes, see the "Multiple Page Sizes" section, chapter 10, "System Performance Tuning" in the *IRIX Admin: System Configuration and Operation* manual.

Summary information is always printed. For an explanation of the limit values, see Table 2-1, page 9.

For additional information, see the `jstat`(1) man page.

Job Limits and Existing IRIX software

The `ps -j` command prints out the process ID, process group ID, session ID, and job ID in hexadecimal:

```
% ps -j
       PID       PGID        SID        JID TTY          TIME CMD
    253430     253430     253430     0x5eac001bd ttyq12  0:00 csh
    254563     254563     253430     0x5eac001bd ttyq12  0:00 ps
```

For additional information, see the `ps`(1) man page.

The array services daemon, `arrayd`(1M), propagates the job ID from the originating machine to any other machines when starting new processes for the job on other machines in a cluster.

For additional information, see the `arrayd`(1M) man page.

The `cpr`(1) command allows you to include job information in the system restart statefile. A JID checkpoint type has been added to the `cpr -p` option. This JID type allows you to checkpoint and restart an entire job. See the example as follows:

```
% cpr -c ckpt02 -p 0x8000000000001234:JID
```

This example checkpoints all the processes contained within a job with the job ID 0x8000000000001234 to the statefile directory `./ckpt02`.

For additional information, see the `cpr`(1) man page.

If you have job limits software installed on your system and want jobs started via the remote shell server (`rshd`(1M)) and remote execution server (`rexecd`(1M)) to recognize the `SIGXCPU` signal, you must update the `/etc/default/rshd` and `/etc/default/rexecd` files, respectively. You must set the `SVR4_SIGNALS` parameter to `NO`. This allows the `rshd` and `rexecd` servers to recognize the `SIGXCPU` signal.

For additional information, see the `rsh`(1M) and `rexecd`(1M) man pages.

Running Job Limits with Message Passing Interface (MPI) Jobs

Message Passing Interface (MPI) jobs requires a great number of file descriptors. By default, a job's current limit for the `files` limit is set to 400 as shown by the `jstat` command with the `-l` option:

```
% jstat -l

JID               OWNER         COMMAND
----------------- ------------- -------------
0x23fc000000000035 user         -csh

LIMIT NAME        USAGE         HIGH USAGE    CURRENT LIMIT  MAX LIMIT
----------------- ------------- ------------- ------------- -------------
cputime           0             0             unlimited     unlimited
datasize          80k           208k          unlimited     unlimited
files             8             28            400           5000
vmemory           2384k         9824k         unlimited     unlimited
ressetsize        608k          2320k         unlimited     unlimited
threads           1             1             2048          2048
processes         2             6             1024          1024
physmem           608k          2320k         unlimited     unlimited
```

If you run MPI jobs on systems with 16 or more CPUs, the default current limit for `files` set at 400 is easily encountered and an error message similar to the following is issued:

```
MPI jobs fail with the error MPI: fork_slaves/fork: Resource temporarily unavailable
MPI: daemon terminated: mice1 - job aborting
```

To avoid this error, set the default current limit for the `files` limit higher when you are running MPI jobs. For information on setting system job limits, see "User Limits Database", page 11 and "Using `systune` to Display and Set Job Limits", page 17.

The following table contains the recommended default current limit for the `files` limit when you are running large MPI jobs depending upon the number of CPUs in your system. The recommended settings are approximate values.

Number of CPUs	**Default Current Limit or Higher**
16	351
17	380
18	410

20	472
25	648
30	848
50	4448

Installing Job Limits

Use the `inst`(1M) software installation tool or the `swmgr`(1M) software management tool to install kernel job limits software. For more information on `inst`(1M) and `swmgr`(1M), see *IRIX Admin: Software Installation and Licensing* in the *IRIX Admin* manual set and their respective man pages.

To install the kernel job limits software on IRIX systems, install this subsystem: `eoe.sw.jlimits`.

Once the job limits software is installed, run the `autoconfig`(1M) command and reboot the system.

Job limits software is only available in the IRIX feature stream.

To turn off job limits, you must deinstall the `eoe.sw.jlimits` software module and then reboot the system.

Troubleshooting Job Limits

If you are updating entries in the ULDB and they do not change the job limit values on your system, make sure that limit names used in the ULDB and limit names used in the `systune` *joblimits* group are exactly the same. The ULDB cannot determine which job limit variables are valid and which are not. If the symbolic names in the ULDB are entered incorrectly, values from the `systune` *joblimits* group will be applied. For information on limit names, see Table 2-1, page 9.

Job Limits 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 job limits software:

User-level man page	**Description**
`jlimit`(1)	Displays and sets resource limits
`jstat`(1)	Displays job status information
`showlimits`(1)	Displays limits information from the user limits database

Administrator Man Pages

The following administrator man page is provided with job limits software:

Administrator man page	**Description**
`genlimits`(1M)	Creates the user limits data base

Application Interface Man Pages

The following online man pages are provided with job limits software to help those who develop applications that use job limits software:

Application interface man page	**Description**
`getjid`(2)	Get job ID
`getjlimit`(2)	Control a job's maximum system resource consumption
`getjusage`(2)	Get job usage information
`killjob`(2)	Terminates all processes for the specified job
`jlimit_startjob`(3c)	Creates a new job
`makenewjob`(2)	Creates a new job container
`setwaitjobpid`(2)	Sets a job to wait for a specified process ID (PID) to call the `waitjob`(2) function

`waitjob`(2)	Obtains information about a terminated job
`uldb_get_limit_values`(3c)	Collection of functions that all interact with the user limits database (ULDB) to retrieve or set limit values for a domain or user.

Error Messages

The following job limits related error messages are returned:

EBUSY	The requested job ID value is in use.
EINVAL	Invalid parameters encountered.
ENOATTR	The domain name or namelist are not specified.
ENOEXIST	The `jlimits` file does not exit.
ENOJOB	A job with the specified job ID cannot be found.
ENOMEM	Sufficient memory is not available.
ENOPKG	The job limits software is not installed.

Miser Batch Processing System

Miser is a resource management facility that provides deterministic batch scheduling of applications with known time and space requirements without requiring static partitioning of system resources. When Miser is given a job, it searches through the time/space pool that it manages to find an allocation that best fits the job's resource requirements.

Miser has an extensive administrative interface that allows most parameters to be modified without requiring a restart. Miser runs as a separate trusted process. All communication to Miser, either from the kernel or the user, is done through a series of Miser commands. Miser accepts requests for process scheduling, process state changes, and batch system configuration control, and returns values and status information for those requests.

This chapter contains the following sections:

- "Miser Overview", page 30
- "Miser Configuration", page 33
- "Miser Configuration Examples", page 39
- "Enabling or Disabling Miser", page 42
- "Submitting Miser Jobs", page 43

Read Me First

The sections in this chapter contain information about installing Miser software on your system. You should reference them in the order they are listed here:

1. For a general description of Miser, see "Miser Overview", page 30.
2. To install the Miser package, see "Enabling or Disabling Miser", page 42.
3. For information on how to configure the Miser queues, see "Miser Configuration", page 33.
4. For information on submitting Miser jobs, see "Submitting Miser Jobs", page 43.
5. For information on Miser man pages, see "Miser Man Pages", page 46.

Miser Overview

Miser manages a set of time/space pools. The time component of the pool defines how far into the future Miser can schedule jobs. The space component of the pool is the set of resources against which a job can be scheduled. This component can vary with time.

A system pool represents the set of resources (number of CPUs and physical memory) that is available to Miser. A set of user-defined pools represents resources against which jobs can be scheduled. The resources owned by the user pools cannot exceed the total resources available to Miser. Resources managed by Miser are available to non-Miser applications when they are unused by a scheduled job.

Associated with each pool is a definition of the pool resources, a set of jobs allocating resources from the pool, and a policy that controls the scheduling of jobs. The collection of the resource pool, jobs scheduled, and policy is called a **queue**.

The queues allow for fine-grained resource management of the batch system. The resources allotted to a queue can vary with time. For example, a queue can be configured to manage 5 CPUs during the day and 20 during the night. The use of multiple queues allows the resources to be partitioned among different users of a batch system. For example, on a 24 CPU system, it is possible to define two queues: one that has 16 CPUs and another that has 6 CPUs (assuming that 2 CPUs have been kept outside the control of Miser). It is possible to restrict access to queues to particular users or groups of users on a system to enforce this resource partition.

The policy defines the way a block of time/space is searched to satisfy the resource request made by the application. Miser has two policies: “default” and “repack.” Default is the first fit policy. Once a job is scheduled, its start and end time remain constant. If an earlier job finishes ahead of schedule, it does not have an effect on the start/end time of future scheduled jobs. On the other hand, in addition to using the first fit policy, repack maintains the order of the scheduled jobs and attempts to reschedule the jobs to pull them ahead in time in the event of a job’s early termination.

Users submit jobs to the queue using the `miser_submit` command, which specifies the queue to which the job should be attached and a resource request to be made against the queue. Each Miser job is an IRIX process group. The resource request is a tuple of time and space. The time is the total CPU wall-clock time if run on a single CPU. The space is the logical number of CPUs and the physical memory required. The request is passed to Miser, and Miser schedules the job against the queue’s resources using the policy attached to the queue. Miser returns a start and end time for the job to the user.

When a job's start time has not yet arrived, the job is in batch state. A job in batch state has lower priority than any non-weightless process. A job in batch state may execute if the system has idle resources; it is said to run opportunistically. When the specified execution time arrives, the job state is changed to batch critical, and the job then has priority over any non-realtime process. The time spent executing in batch state does not count against the time that has been requested and scheduled. While the process is in batch critical state, it is guaranteed the physical memory and CPUs that it requested. The process is terminated if it exceeds its time allotment or uses more physical memory than it had requested.

A job with the static flag specified that was scheduled with the default policy will only run when the segment is scheduled to run. It will not run earlier even if idle resources are available to the job. If a job is scheduled with the repack policy, it may run earlier.

About Logical Number of CPUs

When a job is scheduled by Miser, it requests that a number of CPUs and some amount of memory be reserved for use by the job. When the time period during which these resources were reserved for the job arrives, Miser reserves specific CPUs and some amount of logical swap space for the job.

There are a number of issues that affect CPU allocation for a job. When a job becomes batch critical, Miser will try to find a dense cluster of nodes. If it fails to find such a cluster, it will assign the threads of the job to any free CPUs that are available. These CPUs may be located at distant parts of the system.

The Effect of Reservation of CPUs on Interactive Processes

The way in which Miser handles the reservation of CPUs is one of its strengths. Miser controls and reserves CPUs based on a logical number, not on physical CPUs. This provides Miser with flexibility in how it controls CPU resources.

Interactive and batch processes that run opportunistically are allowed to use all CPUs in a system that have not been reserved for Miser jobs. If new jobs are submitted, Miser attempts to schedule the jobs based on the amount of logical resources still available to Miser. As a result, CPUs could become reserved by Miser, and the interactive processes would no longer be able to execute on the newly reserved CPUs. However, if a resource is not being used by Miser, the resource is free to be used by any other application. Miser will claim the resource when it needs it.

About Miser Memory Management

While Miser only reserves CPUs when they are needed, memory must be reserved before it is needed.

When Miser is started, it is told the number of CPUs and amount of memory that it will be able to reserve for use by jobs. The number of CPUs is a logical number. When a Miser job becomes batch critical, it is assigned a set of CPUs. Until a Miser job requires a CPU (in other words, until a process or thread is ready to run), the CPU is available to the rest of the system. When a Miser job's thread begins executing, the currently non-Miser thread is preempted and resumes on a CPU where no Miser thread is currently running.

Memory resources are quite different than CPU resources. The memory that Miser uses to reserve for jobs is called **logical swap space**. Logical swap space is defined as the sum total of physical memory (less space occupied by the kernel) and all the swap devices.

When Miser begins, it needs to reserve memory for its jobs. However, it does not need to reserve physical memory; it simply needs to make sure that there is enough physical memory plus swap to move non-Miser jobs memory to. Miser does this by reserving logical swap equal to the memory that it requires.

Only jobs that are submitted to Miser are able to use allocations of the logical swap space that was reserved for Miser. However, any physical memory that is not being used by Miser is free to be used by any other application. Miser will claim the physical memory when it needs it.

How Miser Management Affects Users

If a user submits a job to Miser, that job will have an allocation of resources reserved for the requested time period. The job will not have to compete for system resources. As a result, the job should complete more quickly and have more stable run-times than it would if run as an interactive job. However, there is a cost. Because Miser is space sharing the resources, the job must wait until its scheduled reservation period before the requested resources will be reserved. Prior to that time, the non-static job may run opportunistically, competing with the interactive workload, but at a lower priority than the interactive workload.

If a user is working interactively, the user will not have full access to all of the system resources. The user's interactive processes will have access to all of the unreserved CPUs on the system, but the processes will only have a limited amount of logical swap space available for memory allocation. The amount of logical swap space available for non-Miser jobs is the amount not reserved by Miser when it was started.

Miser Configuration

The central configurable aspect of Miser is the set of queues. The Miser queues define the resources allocated to Miser.

The configuration of Miser consists of the following:

- Set up the Miser system queue definition file. Every Miser system must have a Miser system queue definition file. This file's vector definition specifies the maximum resources available to any other queue's vector definition.
- Define the queues by setting up the Miser user queue definition file.
- Enumerate all the queues that will be part of the Miser system by setting up the Miser configuration file.
- Set up the Miser commandline options file to define the maximum CPUs and memory that can be managed by Miser.

Setting Up the Miser System Queue Definition File

The Miser system queue definition file (`/etc/miser_system.conf`) defines the resources managed by the system pool. This file defines the maximum duration of the pool. All other queues must be less than or equal to the system queue. The system queue identifies the maximum limit for resources that a job can request. It is required that a Miser system queue be configured.

Valid tokens are as follows:

POLICY *name*	The policy is always "none" as the system queue has no policy.
QUANTUM *time*	The size of the quantum. A **quantum** is the Miser term for an arbitrary number of seconds. The quantum is used to specify how you want to break up the time/space pool. It is specified in both the system queue definition file and in the user queue definition file and must be the same in both files.
NSEG *number*	The number of resource segments.

SEGMENT	Defines the beginning of a new segment of the vector definition. Each new segment must begin with the *SEGMENT* token. Each segment must contain at a minimum the number of CPUs, memory, and wall-clock time.
START *number*	The number of quanta from 0 that the segment begins at. The origin for time is 00:00 Thursday, January 1st 1970 local time. Miser maps the start and end times to the current time by repeating the queue forward until the current day. For example, a 24-hour queue always begins at midnight of the current day.
END *number*	The number of quanta from 0 that the segment ends at.
NCPUS *number*	The number of CPUs.
MEMORY *amount*	The amount of memory, specified by an integer followed by an optional unit of k for kilobytes, m for megabytes, or g for gigabytes. If no unit is specified, the default is bytes.

The following system queue definition file defines a queue that has a quantum of 20 seconds and 1 element in the vector definition. The start and end times of each multiple are specified in quanta, not in seconds.

The segment defines a resource multiple beginning at 00:00 and ending at 00:20, with 1 CPU and 5 megabytes of memory.

```
POLICY none # System queue has no policy
QUANTUM 20 # Default quantum set to 20 seconds
NSEG 1

SEGMENT
START 0
END 60# Number of quanta (20min*60sec) / 20
NCPUS 1
MEMORY 5m
```

Setting Up the Miser User Queue Definition FIle

The Miser user queue definition file (`/etc/miser_default.conf`) defines the CPUs, the physical memory, the policy name, and the resource pool of the queue. The file consists of a header that specifies the policy of the queue, the number of resource segments, and the quantum used by the queue.

Access to a queue is controlled by the file permissions of the queue definition file. Read permission allows a user to examine the contents of the queue using the `miser_qinfo` command. Execute permission allows a user to schedule a job on a queue using the `miser_submit` command. Write permission allows a user to modify the resources of a queue using the `miser_move` and `miser_reset` commands.

The default user queue definition file can be used as a template for other user queue definition files. Each Miser queue has a separate queue definition file, which is named in the overall Miser configuration file (`/etc/miser.conf`).

Users schedule against the resources managed by the user queues, not against the system queue. If the duration specified by a user queue is less than that specified by the system queue, the user queue will be repeated again and again (for example, the system queue specifies one week and the user queue specifies 24 hours). If the user queue does not divide into the system queue (for example, the system queue is 6 and the user queue is 5), the user queue will repeat evenly.

Valid tokens are as follows:

POLICY *name*	The name of the policy that will be used to schedule applications submitted to the queue. The two valid policies are "default" and repack." Default is the first fit policy; it specifies that once a job is scheduled, its start and end time remain constant. Repack maintains the order of the scheduled jobs and attempts to reschedule the jobs to pull them ahead in time in the event of a job's early termination. Note that both policies initially use the first fit method when scheduling a job.
QUANTUM *time*	The size of the quantum. A **quantum** is the Miser term for an arbitrary number of seconds. The quantum is used to specify how you want to break up the time/space pool. It is specified in both the system queue definition file and in the user queue definition file and must be the same in both files.
NSEG *number*	The number of resource segments.

SEGMENT	Defines the beginning of a new segment of the vector definition. Each new segment must begin with the *SEGMENT* token. Each segment must contain at a minimum the number of CPUs, memory, and wall-clock time.
START *number*	The number of quanta from 0 that the segment begins at. The origin for time is 00:00 Thursday, January 1st 1970 local time. Miser maps the start and end times to the current time by repeating the queue forward until the current day. For example, a 24-hour queue always begins at midnight of the current day.
END *number*	The number of quanta from 0 that the segment ends at.
NCPUS *number*	The number of CPUs.
MEMORY *amount*	The amount of memory, specified by an integer followed by an optional unit of k for kilobytes, m for megabytes, or g for gigabytes. If no unit is specified, the default is bytes.

The following user queue definition file defines a queue using the policy named “default”. It has a quantum of 20 seconds and 3 elements to the vector definition. The start and end times of each multiple are specified in quanta, not in seconds.

- The first segment defines a resource multiple beginning at 00:00 and ending at 00:50, with 50 CPUs and 100 MB of memory.
- The second segment defines a resource multiple beginning at 00:51.67 and ending at 01:00, with 50 CPUs and 100 MB.
- The third segment defines a resource multiple beginning at 01:02.00 and ending at 01:03.33, also with 50 CPUs and 100 MB of memory.

```
POLICY default
QUANTUM 20
NSEG 3

SEGMENT
START 0
END 150 (50min*60sec) / 20
NCPUS 50
MEMORY 100m
```

```
SEGMENT
START 155 ((51min*60sec)+67) / 20
END 185 (1h*60min*60sec) / 20
NCPUS 50
MEMORY 100m

SEGMENT
START 186 ((1h*60min*60sec)+(2min*60sec)) / 20
END 190 ((1h*60min*60sec)+(3min*60sec)+33sec) / 20
NCPUS 50
MEMORY 100m
```

Setting Up the Miser Configuration FIle

The Miser configuration file (`/etc/miser.conf`) lists the names of all Miser queues and the path name of the queue definition file for each queue. This file enumerates all the queue names and their queue definition files.

Every Miser configuration file must include as one of the queues the Miser system queue that defines the resources of the system pool. The Miser system queue is identified by the queue name "system."

Valid tokens are as follows:

QUEUE *queue_name queue_definition_file_path*

> The *queue_name* identifies the queue when using any interface to Miser. The queue name must be between 1 and 8 characters long. The queue name "system" is used to designate the Miser system queue.

The following is a sample Miser configuration file:

```
# Miser config file
QUEUE system /hosts/foobar/usr/local/data/system.conf
QUEUE user /hosts/foobar/usr/local/data/usr.conf
```

Setting Up the Miser CommandLine Options File

The Miser commandline options file (`/etc/config/miser.options`) defines the maximum CPUs and memory that can be managed by Miser.

The `-c` flag defines the maximum number of CPUs that Miser can use. This value is the maximum number of CPUs that any resource segment of the system queue can reserve.

The `-m` flag defines the maximum memory that Miser can use. This value is the maximum memory that any resource segment of the system queue can reserve. The memory reserved for Miser comes from physical memory. The amount of memory that Miser uses should be less than the total physical memory, leaving enough memory for kernel use. Also, the system should have at least the amount of swap space configured for Miser so that if Miser memory is in full use, the system will have enough swap space to move previous non Miser submitted processes out of the way.

The following example sets the `-c` and `-m` values in the commandline options file to *1* and *5* megabytes, respectively:

```
-f/etc/miser.conf -v -d -c 1 -m 5m
```

The `-v` flag specifies verbose mode, which results in additional output.

The `-d` flag specifies debug mode. When this mode is specified, the application does not relinquish control of the tty (that is, it does not become a daemon). This mode is useful in conjunction with the `-v` flag to figure out why Miser may not be starting up correctly.

Note: The `-C` flag can be used to release any Miser reserved resources after the Miser daemon is killed and before it is restarted. For additional information, see the `miser`(1) man page.

Configuration Recommendations

The configuration of Miser is site dependent. The following guidelines may be helpful:

- The system must be balanced for interactive/batch use. One suggestion is to keep at least one or two processors outside the control of Miser at all times. These two processors will act as the interactive portion of the system when all of the Miser managed CPUs are reserved. For an interactive load, you typically want the load average for the CPUs to be less than 2.0. Keep this in mind as you adjust for the optimal number of free CPUs.
- The amount of free logical swap should be balanced against the number of free CPUs. When you have a system with *N* CPUs, you should also have an appropriate amount of memory to be used by processes running on those *N* CPUs.

Also, many system administrators like to back up this memory with swap space. If you think of the free CPUs as a separate system and provide memory and swap space accordingly, interactive work should perform well. Remember that the free memory not reserved by Miser is logical swap space (the combination of physical memory and the swap devices).

- Be careful when using virtual swap. When no Miser application is running, time-share processes can consume all of physical memory. When Miser runs, it begins to reclaim physical memory and swaps out time-share processes. If the system is using virtual swap, there may be no physical swap to move the process to, and at that point the time-share process may be terminated.

Miser Configuration Examples

In the examples used in this section, the system has 12 CPUs and 160 MB available to user programs.

Example 1:

In this example, the system is dedicated to batch scheduling with one queue, 24 hours a day.

The first step is to define a system queue. You must decide how long you want the system queue to be. The length of the system queue defines the maximum duration of any job submitted to the system. For this system, you have determined that the maximum duration for any one job can be 48 hours, so you define the system vector to have a duration of 48 hours.

```
# The system queue /usr/local/miser/system.conf
POLICY none # System queue has no policy
QUANTUM 20 # Default quantum set to 20 seconds
NSEG 1

SEGMENT
NCPUS 12
MEMORY 160m
START 0
END 8640 # Number of quanta (48h*60 min*60 sec) / 20
```

The next step is to define a user queue.

```
# The user queue /usr/local/miser/physics.conf
POLICY default # First fit, once scheduled maintains start/end time
QUANTUM 20 # Default quantum set to 20 seconds
NSEG 1

SEGMENT
NCPUS 12
MEMORY 160m
START 0
END 8640 # Number of quanta (48h*60 min*60 sec) / 20
```

The last step is to define a Miser configuration file:

```
# Miser config file
QUEUE system /usr/local/miser/system.conf
QUEUE physics /usr/local/miser/physics.conf
```

Example 2:

In the following example, the system is dedicated to batch scheduling, 24 hours a day, and split between two user groups: chemistry and physics. The system must be divided between them with a ratio of 75% for physics and 25% for chemistry.

The system queue is identical to the one given in Example 1.

The physics user queue appears as follows:

```
# The physics queue /usr/local/miser/physics
POLICY default # System queue has no policy
QUANTUM 20 # Default quantum set to 20 seconds
NSEG 1

SEGMENT
NCPUS 8
MEMORY 120m
START 0
END 8640 # Number of quanta (48h*60min*60sec) / 20
```

Next, you define the chemistry queue:

```
# The chemistry queue /usr/local/miser/chemistry.conf
POLICY default # System queue has no policy
QUANTUM 20 # Default quantum set to 20 seconds
NSEG 1
```

```
SEGMENT
NCPUS 4
MEMORY 40m
START 0
END 8640 # Number of quanta (48h*60min*60sec) / 20
```

To restrict access to each queue, you create the user group physics and the user group chemistry. You then set the permissions on the physics queue definition file to execute only for group physics and similarly for the chemistry queue.

Having defined the physics and chemistry queue, you can now define the Miser configuration file:

```
# Miser configuration file
QUEUE system /usr/local/miser/system.conf
QUEUE physics /usr/local/miser/physics.conf
QUEUE chem /usr/local/miser/chemistry.conf
```

Example 3:

In this example, the system is dedicated to time-sharing in the morning and to batch use in the evening. The evening is 8:00 P.M. to 4:00 A.M., and the morning is 4:00 A.M. to 8:00 P.M.

First you define the system queue.

```
# The system queue /hosts/foobar/usr/local/data/system.conf
POLICY none # System queue has no policy
QUANTUM 20 # Default quantum set to 20 seconds
NSEG 2

SEGMENT
NCPUS 12
MEMORY 160m
START 0
END 720 # (4h*60min*60sec) / 20

SEGMENT
NCPUS 12
MEMORY 160m
START 3600 # (8pm is 20 hours from UTC, so 20h*60min*60sec) / 20
END 4320
```

Next, you define the batch queue:

```
# User queue
POLICY repack # Repacks jobs (FIFO) if a job finishes early
QUANTUM 20 # Default quantum set to 20 seconds
NSEG 2

SEGMENT
NCPUS 12
MEMORY 160m
START 0
END 720 # (4h*60min*60sec) / 20

SEGMENT
NCPUS 12
MEMORY 160m
START 3600 # (8pm is 20 hours from 0, so 20h*60min*60sec) / 20
END 4320
```

The last step is to define a Miser configuration file:

```
# Miser config file
QUEUE system /usr/local/miser/system.conf
QUEUE user /usr/local/miser/usr.conf
```

Enabling or Disabling Miser

The following steps are required to set up the Miser batch processing system:

1. Use the `inst`(1M) utility to install the `eoe.sw.miser` subsystem from your IRIX distribution media.
2. Modify the Miser configuration files as appropriate for your site. For information on the Miser configuration files, see "Miser Configuration Examples", page 39.

   After the Miser configuration files are modified appropriately, Miser can be selected for boot-time startup with the `chkconfig`(1) command and the system can be rebooted, or Miser can be started directly by **root** with the command `/etc/init.d/miser start`. When starting Miser manually without rebooting, the `chkconfig` command must be issued first or Miser will not start up.

3. To enable Miser manually, use the following command sequence:

   ```
   chkconfig miser on
   /etc/init.d/miser start
   ```

4. Miser can be stopped at any time by **root**. To disable Miser, use the following command sequence:

   ```
   /etc/init.d/miser stop
   /etc/init.d/miser cleanup
   ```

Running Miser jobs are not stopped, and the current committed resources cannot be reclaimed until the jobs are terminated. If you are going to restart Miser after stopping it, you do not need to run the `miser cleanup` command.

Note: The Miser `-C` flag can be used to release any Miser reserved resources after the Miser daemon is killed and before it is restarted.

Submitting Miser Jobs

The command to submit a job so that it is managed by Miser is as follows:

```
miser_submit -q queue -o c=cpus,m=memory, t=time[,static] command
miser_submit -q queue -f file command
```

`-q` *queue*	Specifies the name of the queue against which to schedule the application.
`-o c=`*cpus*`,m=`*memory*`, t=`*time*`[,static]`	Specifies a block of resources. The CPUs must be an integer up to the maximum number of CPUs available to the queue being scheduled against. The memory consists of an integer followed by a unit of **k** for kilobyte, **m** for megabyte, or **g** for gigabyte. If no unit is specified, the default is bytes. Time can be specified either as an integer followed by a unit specifier of **h** for hours, **m** for minutes, or **s** for seconds, or by a string in the format *hh:mm:ss*. A job with the `static` flag specified that was scheduled with the default policy will only run when the segment is scheduled to run. It will not run earlier

	even if idle resources are available to the job. If a job is scheduled with the repack policy, it may run earlier.
`-f` *file*	File that specifies a list of resource segments. This flag allows greater control over the scheduling parameters of a job.
command	Specifies a script or program name.

For additional information, see the `miser_submit`(1) and `miser_submit`(4) man pages.

Querying Miser About Job Schedule/Description

The command to query Miser about the schedule/description of a submitted job is as follows:

```
miser_jinfo -j bid [-d]
```

The *bid* is the ID of the Miser job and is the process group ID of the job. The `-d` flag prints the job description including job owner and command.

Note that when the system is being used heavily, Miser swapping can take some time. Therefore, the Miser job may not begin processing immediately after it is submitted.

For additional information, see the `miser_jinfo`(1) man page.

Querying Miser About Queues

The command to query Miser for information on Miser queues, queue resource status, and a list of jobs scheduled against a queue is as follows:

```
miser_qinfo -Q|-q queue [-j]|-a
```

The `-Q` flag returns a list of currently configured Miser queue names. The `-q` flag returns the free resources associated with the specified queue name. The `-j` flag returns the list of jobs currently scheduled against the queue. The `-a` flag returns a list of all scheduled jobs, ordered by job ID, in all configured Miser queues and also produces a brief description of the job.

For additional information, see the `miser_qinfo`(1) man page.

Moving a Block of Resources

The command to move a block of resources from one queue to another is as follows:

```
miser_move -s srcq -d destq -f file
miser_move -s srcq -d destq -o s=start,e=end,c=CPUs,m=memory
```

This command removes a tuple of space from the source queue's vector and adds it to the destination queue's vector, beginning at the start time and ending at the end time. The resources added or removed do not change the vector definition, and are, therefore, temporary. The command returns a table that lists the start and end times of each resource transfer and the amount of resources transferred.

The `-s` and `-d` flags specify the names of any valid Miser queues. The `-f` flag contains a resource block specification. The `-o` flag specifies a block of resources to be moved. The start and end times are relative to the current time. The CPUs are an integer up to the maximum free CPUs associated with a queue. The memory is an integer with an identifier of **k** for kilobyte, **m** for megabyte, or **g** for gigabyte.

Note: The resource transfer is temporary. If Miser is killed or crashes, the resources transferred are lost, and Miser will be unable to restart.

For additional information, see the `miser_move`(1) and `miser_move`(4) man pages.

Resetting Miser

The command to reset Miser with a new configuration file is as follows:

```
miser_reset -f file
```

This command forces a running version of Miser to use a new configuration file (specified by `-f` *file*). The new configuration will succeed only if all scheduled jobs can be successfully scheduled against the new configuration.

For additional information, see the `miser_reset`(1) man page.

Terminating a Miser Job

The `miser_kill` command is used to terminate a job submitted to Miser. This command both terminates the process and contacts the Miser daemon to free any resources currently committed to the submitted process. For additional information, see the `miser_kill`(1) man page.

Miser and Batch Management Systems

This section discusses the differences between a Miser job and a batch job from a batch management system such as the Network Queuing Environment (NQE) or Load Share Facility (LSF).

Miser and batch management systems such as NQE each lack certain key characteristics. For Miser, these characteristics are features to protect and manage the Miser session. For batch management systems, the ability to guarantee resources is lacking. However, these two systems used together provide a much more capable solution, provided the batch management system supports the Miser scheduler.

If your site does not need the job management and protection provided by a batch management system, then Miser alone may be an adequate batch system. However, most production-quality environments require the support and protection provided by batch systems such as NQE or LSF. These sites should run a batch management system in cooperation with the Miser scheduler.

Miser 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 Miser software:

User-level man page	Description
`miser`(1)	Miser resource manager; starts the miser daemon.
`miser_jinfo`(1)	Queries Miser about the schedule and description of a submitted job.
`miser_kill`(1)	Kills a Miser job.

`miser_move`(1)	Moves a block of resources from one queue to another.
`miser_qinfo`(1)	Queries information on miser queues, queue resource status, and list of jobs scheduled against a queue.
`miser_reset`(1)	Resets miser with a new configuration file.
`miser_submit`(1)	Submits a job to a miser queue.

File Format Man Pages

The following file format descriptions man pages are provided with Miser software:

File Format man page	Description
`miser`(4)	Miser configuration files
`miser_move`(4)	Miser resource transfer list
`miser_submit`(4)	Miser resource schedule list

Miscellaneous Man Pages

The following miscellaneous man pages are provided with Miser software:

Miscellaneous man page	Description
`miser`(5)	Miser Resource Manager overview

Chapter 4

Cpuset System

A **cpuset** is a named set of CPUs, which may be defined to be restricted or open. A restricted cpuset allows only processes that are members of the cpuset to run on the set of CPUs. An open cpuset allows any process to run on its CPUs, but a process that is a member of the cpuset can run only on the CPUs belonging to the cpuset. A cpuset is defined by a cpuset configuration file and a name.

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. Cpusets may optionally restrict both kernel and user memory.

When the memory restriction feature is enabled, a set of nodes each containing a set of CPUs is computed from the list of CPUs supplied and memory allocations can be limited to the CPUs assigned to the nodes. Allocation limits can be restricted to the available physical memory or overflow can be swapped to the swap file.

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 affect these processes have on the rest of the system is minimized.

A system administrator might want to restrict normal system usage of a large system to part of the machine and use the rest of the system for special purposes. The `boot_cpuset`(4) tool provides a method to restrict all normal start-up processes (including `init`, `inetd`, and so on) to some portion of the machine and allow specific users to use the other portion of the machine for their special purpose applications. The kernel maintains strict processor and memory separation between the two system portions. An administrator, for example, might choose to divide a system into two halves with one half supporting normal system usage and the other half dedicated to a particular application. The advantage this mechanism has over physical reconfiguration is that the configuration may be changed with a simple robot and does not need to be aligned on a hardware module boundary.

The `cpuset -q` *cpuset_name* `-p` command allows you to see the properties of particular cpuset, such as, the number of processes and CPUs associated with the specified cpuset. For more information on cpuset properties, see "Obtaining the Properties Associated with a Cpuset", page 64 and the `cpuset`(1) man page.

Cpusets can be used in conjunction with a batch processing system, like the Load Sharing Facility (LSF) or Portable Batch System (PBS), for data center resource management to improve the performance of large applications.

For more information on dividing a system, see Chapter 4, “Configuring the IRIX Operating System” in the *IRIX Admin: System Configuration and Operation* manual.

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:

Using Cpusets

This section describes the basic steps for using cpusets and the `cpuset`(1) command. For a detailed example, see "Cpuset System Tutorial", page 53.

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

To use cpusets, perform the following steps:

1. Create a cpuset configuration file and give it a name. For the format of this file, see "Cpuset Configuration File", page 59. For restrictions that apply to CPUs belonging to cpusets, see "Restrictions on CPUs within Cpusets", page 52.
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 [-A command]|[-c -f filename]|[-d]|[-l][-m]|[-Q]|[-p]] | -C | -Q | -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.
`-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.
`-q` *cpuset_name* `-d`	Destroys the specified cpuset. A cpuset can only be destroyed if there are no processes currently attached to it.
`-q` *cpuset_name* `-l`	Lists all the processes in the cpuset.
`-q` *cpuset_name* `-m`	Moves all the attached processes out of the cpuset.
`-q` *cpuset_name* `-Q`	Prints a list of the CPUs that belong to the cpuset.

`-q` *cpuset_name* `-p`	Prints out the permissions, ACLs, MAC labels, flags, number of processes, and the CPUs associated with the specified 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 52, and "Cpuset System Tutorial", page 53.

Restrictions on CPUs within Cpusets

The following restrictions apply to CPUs belonging to cpusets:

- A CPU can belong to only one cpuset.
- CPU 0 cannot belong to an `EXCLUSIVE` cpuset.
- A CPU cannot be both restricted or isolated (see `mpadmin`(1) and `sysmp`(2)) and also be a member of a 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.

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

Cpuset System Tutorial

This section gives a detailed example of how to divide a system using cpusets. It contains a simple procedure to follow to divide the example system into cpusets with references to additional explanatory information.

Figure 4-1, page 54, shows a block diagram of a system with 16 processors with three cpusets. This section provides examples of configuration files and the commands used to create a boot cpuset containing half of the system's CPUs for normal system usage, and two cpusets named Green and Blue, respectively, for specified purposes. The Green cpuset specifies a closed cpuset restricted to a specific application to be executed by members of group `artists`. The Blue cpuset specifies a second closed cpuset restricted to a specific application to be executed my members of group `writers`.

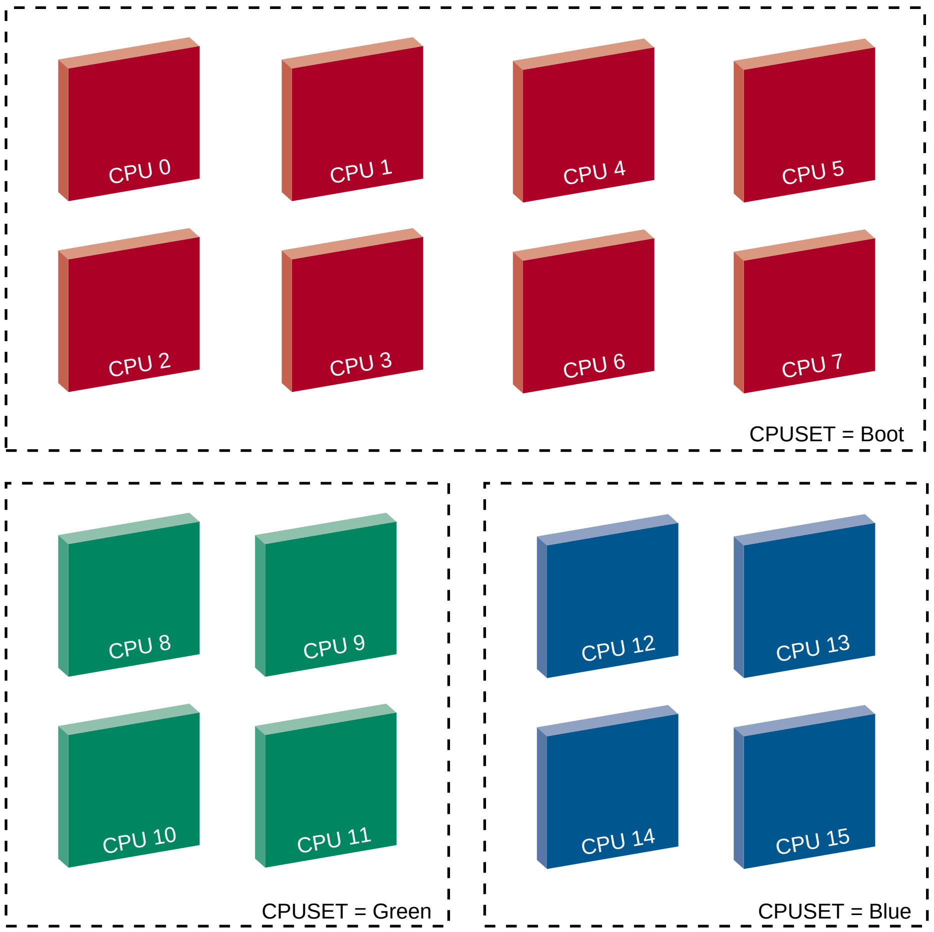

Figure 4-1 Dividing a System Using Cpusets

Perform the following steps to divide a system with 16 processors into 3 cpusets as shown in Figure 4-1, page 54:

1. Create a file named `boot_cpuset.config` to create a boot cpuset and divide half of a 16 CPU system dedicated to normal system usage. The boot cpuset

contains all standard processes on the system such as daemons, interactive or background processing, scripts, and so on. The contents of this file are as follows:

```
# boot
MEMORY_LOCAL
MEMORY_MANDATORY

CPU 0
CPU 1
CPU 2
CPU 3
CPU 4
CPU 5
CPU 6
CPU 7
```

For information on the `boot_cpuset.config` file, see "Boot Cpuset", page 57.

2. Use the `chkconfig`(1M) command with the `-f` option to create the `/etc/config/boot_cpuset` file that contains the following:

   ```
   chkconfig boot_cpuset on
   ```

   For information on the `/etc/config/boot_cpuset` file, see "Boot Cpuset", page 57.

   For an explanation of the `MEMORY_LOCAL` and `MEMORY_MANDATORY` flags, see "Cpuset Configuration File", page 59.

   When the system is rebooted, the boot cpuset will be created.

3. Create a dedicated cpuset called `Green` and assign a specific application, in this case, MovieMaker 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
      EXCLUSIVE
      MEMORY_LOCAL
      MEMORY_MANDATORY

      CPU 8
      ```

```
CPU 9
CPU 10
CPU 11
```

For an explanation of the EXCLUSIVE, MEMORY_LOCAL, and MEMORY_MANDATORY flags, see "Cpuset Configuration File", page 59.

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 moviemaker on the Green cpuset.

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

```
cpuset -q Green -f cpuset_1
```

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

```
cpuset -q Green -A moviemaker
```

For more information on the cpuset(1) command, see "cpuset Command", page 59.

The moviemaker job threads will only run on CPUs in this cpuset. MovieMaker 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.

4. Create a third cpuset file called Blue and specify an application that will run only on the 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
MEMORY_MANDATORY

CPU 12
CPU 13
CPU 14
CPU 15
```

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

c. Create the cpuset with the configuration file specified by the `-f` parameter and the name specified by the `-q` parameter.

```
cpuset -q Blue -f cpuset_2
```

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

```
cpuset -q Blue -A bookmaker
```

The `bookmaker` job threads will only run on this cpuset. BookMaker 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.

Boot Cpuset

The `boot_cpuset.so`(4) library provides a method for containing the `init`(1M) process and all of its descendents within a cpuset. Because all standard processes are descendents of the `init` process, this means that all standard processes on the system such as daemons, interactive or background processing, scripts, and so on, are confined to this cpuset. This cpuset is named **`boot`**.

Note: The `boot_cpuset.so` library is provided only on SGI 2000 and SGI 3000 series of systems, that is, systems that are based on ccNUMA architecture.

The `boot_cpuset.so` library is located in the `/lib32` directory and its behavior is controlled by the following files:

- `/etc/config/boot_cpuset`
- `/etc/config/boot_cpuset.config`

The `/etc/config/boot_cpuset` file follows the `chkconfig`(1M) command convention and contains a line similar to the following:

```
chkconfig boot_cpuset on
```

You can use the `chkconfig`(1M) command to configure the `boot_cpuset.so`(4) library on or off. If the library is configured on by `init` during system start up, the `boot_cpuset.so` library is loaded and executed and the cpuset is created. If the library is configured off, the library will exit and `init` will resume normal processing.

The `/etc/config/boot_cpuset.config` file is the configuration file specifying the cpuset. It follows the same conventions as the `cpuset`(4) configuration file.

The following example shows a `boot_cpuset.config` file that would divide half of an eight CPU system for normal system usage:

```
# the boot_cpuset
MEMORY_LOCAL
MEMORY_MANDATORY

CPU 0
CPU 1
CPU 2
CPU 3
```

Note: CPU 0 cannot belong to an `EXCLUSIVE` cpuset. For restrictions that apply to CPUs belonging to cpusets, see "Restrictions on CPUs within Cpusets", page 52.

The second configuration file shows a cpuset that could be dedicated to a specific application:

```
# the cpuset dedicated to a specific application
EXCLUSIVE
MEMORY_LOCAL
MEMORY_MANDATORY

CPU 4
CPU 5
CPU 6
CPU 7
```

For more information see "Cpuset Command and Configuration File", page 59 and the `cpuset`(4) man page.

Cpuset Command and Configuration File

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

`cpuset` Command

The `cpuset`(1) command is used to define and manage a set of CPUs called a **cpuset**. A cpuset is a named set of CPUs, which may be defined as restricted or open. The `cpuset` command creates and destroys cpusets, retrieves information about existing cpusets, and attaches a process to a cpuset. Attachment to a cpuset is inherited across the `fork`(2) system call. Consequently, all processes that are children of an attached process will also be attached to the same cpuset.

Note: The `cpuset` command does not require the use of the Miser batch processing system.

A restricted cpuset allows only processes that are attached to the cpuset to run on the set of CPUs. An open cpuset allows any process to run on its CPUs, but a process that is a attached to the cpuset can only run on the CPUs belonging to the cpuset.

For the SGI 2000 and SGI 3000 series of systems, that is, systems that are based on ccNUMA architecture, the administrator can restrict memory allocation to the nodes that contain the CPUs defined in the cpuset. For more information, see `MEMORY_MANDATORY` flag description that follows and the `cpuset`(4) man page.

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 three and eight characters long; names of two or fewer characters are reserved. Each cpuset on your system must have a separate cpuset configuration file.

Note: In a 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` (see `mount`(1M) command fails.

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.

By convention, CPU numbering on SGI systems vary between zero and the number of processors on the system minus one. The `mpadmin -n` command reports which processors are physically configured on a system. You can also use the `hinv -vm` command to show the hardware configuration of your system. For more information on the CPU naming convention and system hardware configuration, see Chapter 4, "Configuring the IRIX Operating System", in the *IRIX Admin: System Configuration and Operation* manual and the `mpadmin`(1) and `hinv`(1) man pages.

The following is a sample configuration file that describes an exclusive cpuset containing 3 CPUs:

```
# cpuset configuration file
EXCLUSIVE
MEMORY_LOCAL
MEMORY_EXCLUSIVE

CPU 1
CPU 5
CPU 10
```

This specification will create a cpuset containing 3 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.

When the `MEMORY_MANDATORY` flag is set, the jobs running on the cpuset can only use memory from nodes containing the CPUs in this cpuset. The `MEMORY_LOCAL` flag is only an advisory while the `MEMORY_MANDATORY` flag is enforced by the kernel.

Note: On a system with both Miser and cpuset configured, conflicts may occur between a CPU that a Miser queue is using and a CPU assigned to a cpuset. Miser does not have access to CPUs that belong to a cpuset configured with the `EXCLUSIVE` flag set. Avoid running Miser and cpusets on the same system.

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 avoids allocating memory from nodes contained in this cpuset. If kernel memory requests cannot be satisfied from outside this cpuset, this option is ignored and allocations occur from within the cpuset. Currently, this

	option prevents only the system buffer cache from being placed on the specified nodes.
	Warning: Most sites running cpusets should **not** use this option. The use of this option can degrade system performance because kernel memory allocations become concentrated on the remaining system nodes. This option is effective only for certain workload patterns and can cause severe performance penalties in other situations. Do not use this option unless it is indicated by SGI support staff.
	This option was introduced in the IRIX 6.5.7 release.
`MEMORY_MANDATORY`	The kernel will limit all memory allocations to nodes that are contained in this cpuset. If memory requests cannot be satisfied, the allocating process will sleep until memory is available. The process will be killed if no more memory can be allocated.
`POLICY_PAGE`	Requires the `MEMORY_MANDATORY` token. This is the default policy if no policy is specified. This policy will cause the kernel to page user pages to the swap file (see `swap`(1M)) to free physical memory on the nodes contained in this cpuset. If swap space is exhausted, the process will be killed.

`POLICY_KILL`	Requires the `MEMORY_MANDATORY` token. 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.

Installing the Cpuset System

Although the Cpuset System is functionally separate from the Miser batch processing system, the current Cpuset System was developed in conjunction with the software development of Miser. The Cpuset System software is contained within the Miser subsystem software. To install the Cpuset System software, see "Enabling or Disabling Miser", page 42.

Obtaining the Properties Associated with a Cpuset

The `cpuset -q` *cpuset_name* `-p` command allows you to see the various properties associated with a particular cpuset as follows:

- Permissions on the configuration file that define access to the cpuset
- Access Control Lists (ACLs)
- Mandatory Access Control (MAC) labels
- Flags such as `MEMORY_EXCLUSIVE`

  For more information on flags associated with a cpuset, see "Cpuset Configuration File", page 59 and the `cpuset`(4) man page.
- Number of processes
- CPUs

The `cpusetGetProperties`(3x) function in the cpuset library is used retrieve various properties of the specified cpuset. The `cpusetFreeProperties`(3x) function is used to release memory used by `cpuset_Properties_t` structure. For more information, see "Retrieval Functions", page 156, and "Clean-up Functions", page 174, and the `cpusetGetProperties`(3x) and `cpusetFreeProperties`(3x) man pages.

Cpuset System and Trusted IRIX

This section describes how to run cpusets in a Trusted IRIX environment.

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.

Read access allows a user to retrieve information about a cpuset while execute permission allows the user to attach a process to the cpuset.

Cpusets on IRIX requires two user classes: root and user. The root class creates, destroys, moves a process, and adds a process to the cpuset. The user class is governed by the file permissions of the configuration file for the given cpuset.

Given a configuration file with the following characteristics:

```
Permissions     Owner   Group   Size    Filename
---------------------------------------------------------

 -rwxr-----      root    cpuset  512     cpuset.test
```

Group read permission allows a user belonging to the group cpuset to list all cpusets in the cpuset defined by the `cpuset.test` file and get a listing of all processes in this cpuset. In order for the user to add processes to the cpuset governed by the `cpuset.test` file, you would need to change the permissions as follows:

```
Permissions     Owner   Group   Size    Filename
---------------------------------------------------------

-rwxr-x---      root    cpuset  512     cpuset.test
```

In a Trusted IRIX environment, permissions are governed by the `/etc/capability` file. See the `capability`(4) and `capabilities`(4) man pages for more information on the capability mechanism that provides fine grained control over the privileges of a process. Each user in the `capability` file has a set of minimum and maximum permissions. Consequently, root does not have any special abilities except to be able to use the `suattr`(1M) call so that it may assume any capabilities and permissions. Capabilities and permissions are also narrowed by the use of mandatory access control (MAC) labels and access control lists (ACLs).

In Trusted IRIX, to allow a user belonging to the group `cpuset` to list all cpusets in the cpuset defined by the `cpuset.test` file and get a listing of all processes in this cpuset, you must perform the following:

- Assign the user with a MAC label of userlow.
- Make the following entry in the `/etc/capability` file: `cpuuser1:all=:all=`

You can **not** assign a user all capabilities with effective, inherited, and permissive rights (`+eip`) added. If you add `+eip`, the user will gain more privileges than allowed by the Cpuset system.

A Trusted IRIX user with a `cpuuser1:all=:all=` entry in the `/etc/capability` file, has the same permissions as the user class in IRIX.

The root class in Trusted IRIX must have the `CAP_SCHED_MGT+eip` capability to create and destroy cpusets and to move process out of the cpuset.

In Trusted IRIX, you can use ACLs to control group permissions. With ACLs, you can easily select which users in the group can add a process to the cpuset. You can use

ACLs to control a user’s access to a cpuset without that user belonging to the group owner of the configuration file.

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 and existing cpuset, and to attach a process and all of its children to a cpuset.

For information on using the Cpuset Library, see "Application Programming Interface for the Cpuset System", page 140.

`cpusetAttachPID` and `cpusetDetachPID`

The new `cpusetAttachPID`(3x) function in the cpuset library allows a specific process, identified by its PID value, to be attached to a cpuset. The new `cpusetDetachPID` function allows a specific process, identified by its PID value, to be detached from a cpuset. The ability to attach and detach specific processes to or from a cpuset is controlled by the permissions of the cpuset configuration file and the ownership of the processes involved. For more information on the cpuset configuration file, see "Cpuset Configuration File", page 59.

Using the `cpusetAttachPID` and `cpusetDetachPID` Functions

The `cpusetAttachPID` and `cpusetDetachPID` functions should not be used with the `MEMORY_MANDATORY` flag set to avoid memory latency problems. Because a cpuset will only use memory from the original compute nodes, use the `cpusetAttachPID` and `cpusetDetachPID` functions as follows:

Figure 4-2, page 67, shows several jobs running in two cpusets each containing four CPUs. A prime job requires a new cpuset using all eight CPUs. To create the new cpuset, perform the following steps:

1. Use the `cpusetDetachPID` function to move all jobs out of cpuset A and cpuset B.
2. Suspend the jobs running on cpuset A and B.
3. Use the `cpusetDestroy`(3x) function to destroy cpuset A and cpuset B.

4. Use the `cpusetCreate`(3x) function to create the new cpuset for the prime job.
5. Run the prime job in the new cpuset.
6. Destroy the new cpuset when the prime job has completed running.
7. Recreate cpuset A and B as exactly before.
8. Restart the suspended jobs.
9. Use the `cpusetAttachPID` function to reattach each job to its respective cpuset.

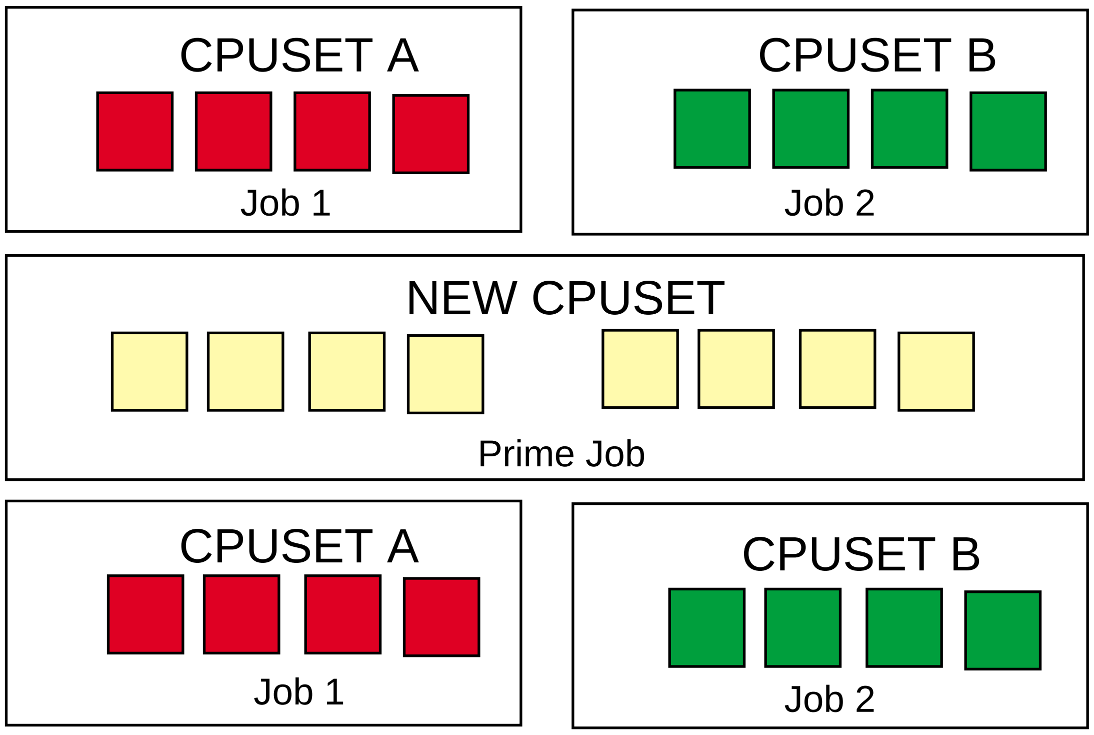

Figure 4-2 Using the `cpusetAttachPID` and `cpusetDetachPID` Functions

Cpuset System Man Pages

The `man` command provides online help on all resource management commands. To view a man page online, type `man` *commandname*. For printed versions of the cpuset library man pages, see "Application Programming Interface for the Cpuset System", page 140 in Appendix A.

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.
`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.
`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.

File Format Man Pages

The following file format descriptions 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.

Chapter 5

Comprehensive System Accounting

The IRIX system has three types of accounting: basic accounting, extended accounting, and Comprehensive System Accounting (CSA). You can use either one type of accounting or a combination of them, depending on your site's accounting needs. This chapter contains detailed information about CSA.

You can use the three types of IRIX accounting to log and charge for certain types of system activity. 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.

Basic accounting consists of standard UNIX accounting features. Basic accounting is process oriented; a new accounting record is produced for each process that has been run, containing statistics about the resources used by that individual process. The `runacct`(1M) command is the main daily accounting shell script usually initiated by `cron`(1M). The `runacct`(1M) command processes accounting records written into the process accounting data file.

Extended accounting is an IRIX feature that has extended process accounting capabilities, along with project and array session accounting features. Unlike basic processing accounting and CSA which write accounting data directly to an accounting data file, extended accounting writes data files using the system audit trail (SAT) facility. Audit data is collected directly from the kernel by the `satd`(1M) program. The extended accounting data is a superset of the data collected and reported by basic accounting.

CSA provides additional capabilities that provide more detailed, accurate accounting data per job. It also contains data from some daemons. The `csarun`(1M) command, usually initiated by the `cron`(1M) command, directs the processing of the CSA daily accounting files. The `csarun`(1M) command processes accounting records written into the CSA accounting data file.

For more detailed information on basic accounting and extended accounting, see "About the Process Accounting System" and "IRIX Extended Accounting", respectively, in Chapter 7, "System Accounting" *IRIX Admin: Backup, Security and Accounting* manual.

This chapter contains the following sections:

Read Me First

The sections in this chapter contain information about installing CSA software on your system. You should reference them in the order they are listed here:

1. For a general description of CSA, see "CSA Overview", page 73.
2. To install the CSA package and job limits package used by CSA, see "Enabling or Disabling CSA", page 76.
3. For information about CSA directories and files, see "CSA Files and Directories", page 77.
4. For detailed information about CSA, such as, setting CSA up on your system, daily operation, tailoring CSA to your system, see "Comprehensive System Accounting Expanded Description", page 84.
5. For a list of CSA man pages, see "CSA Man Pages", page 127.
6. For information about the types of reports you can generate using CSA, see "CSA Reports", page 119.

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 (tape, NQS and workload management systems)
- 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 NQS jobs or 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 NQS identifier or the workload management identifier.

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. An NQS or 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 105.

The CSA accounting records are not written into the basic accounting `pacct` file but are written into a separate CSA `/var/adm/acct/day/pacct` file. The CSA commands can only be used with CSA generated accounting records. Similarly, the basic accounting commands can only be used with the records generated by basic accounting.

There are four user exits available with the `csarun`(1M) daily accounting script. There is one user exit available with the `csaperiod`(1M) 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`(1M) and `csaperiod`(1M) man pages for further information.

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 `csacom` command is very similar to the basic accounting `acctcom`(1) command. 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.

Like any accounting or monitoring package, the CSA features do contribute to overall system overhead. For this reason, CSA is disabled in the kernel by default. To enable CSA, see "Enabling or Disabling CSA", page 76.

Concepts and Terminology

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

Term	**Description**
Daily accounting	Daily accounting is the processing, organizing, and reporting of the raw accounting data, generally performed once per day. In basic accounting, daily accounting can only be run once a day. With CSA, it 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). 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-NQS or non-workload management jobs, a job consists of all accounting data for a given job ID during a single boot period. An NQS job consists of the accounting data for all job IDs associated with the job's NQS sequence number, and a workload management job consists of the accounting data for all job IDs associated with the workload management request ID. NQS or 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 NQS or workload management jobs have multiple job IDs. CSA treats all phases of an NQS job or workload management job as being in the same 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. In basic accounting, this refers to accounting that is run on a monthly basis. CSA, however, 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/adm/acct/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:

1. Use the `inst`(1M) utility to install the `eoe.sw.csaacct` subsystem from your IRIX distribution media. Installing CSA also requires that the `eoe.sw.acct` and `eoe.sw.jlimits` subsystems are installed.

2. Enable CSA within the kernel by using the `systune`(1M) utility to set `do_csaacct` to a nonzero value. It will be necessary to reboot the system after completing this step.

3. Configure CSA on across system reboots by using the `chkconfig`(1M) utility as follows:

   ```
   chkconfig csaacct on
   ```

4. Modify the CSA configuration variables in `/etc/csa.conf` as desired.

5. Use the `csaswitch`(1M) command to configure on the accounting record types and thresholds defined in `/etc/csa.conf` as follows:

   ```
   csaswitch -c on
   ```

   This step will be done automatically for subsequent system reboots when CSA is configured on via the `chkconfig`(1M) utility.

   For information on adding entries to the `crontabs` file so that the `cron`(1M) command automatically runs daily accounting, see "Setting Up CSA", page 86.

The following steps are required to disable CSA job accounting:

1. To turn off CSA, use the `csaswitch`(1M) command:

   ```
   csaswitch -c halt
   ```

2. To stop CSA from initiating after a system reboot, use the `chkconfig`(1M) command:

   ```
   chkconfig csaacct off
   ```

3. Disable CSA within the kernel by using the `systune`(1M) utility to set `do_csaacct` to a zero value. It will be necessary to reboot the system after completing this step.

CSA Files and Directories

The following sections describe the CSA files and directories.

Files in the `/var/adm/acct` Directory

The `/var/adm/acct` directory contains CSA data and report files within various subdirectories. `/var/adm/acct` contains data collection files used by CSA. CSA and IRIX basic accounting access separate `pacct` files. The following diagram shows the directory and file layout for CSA:

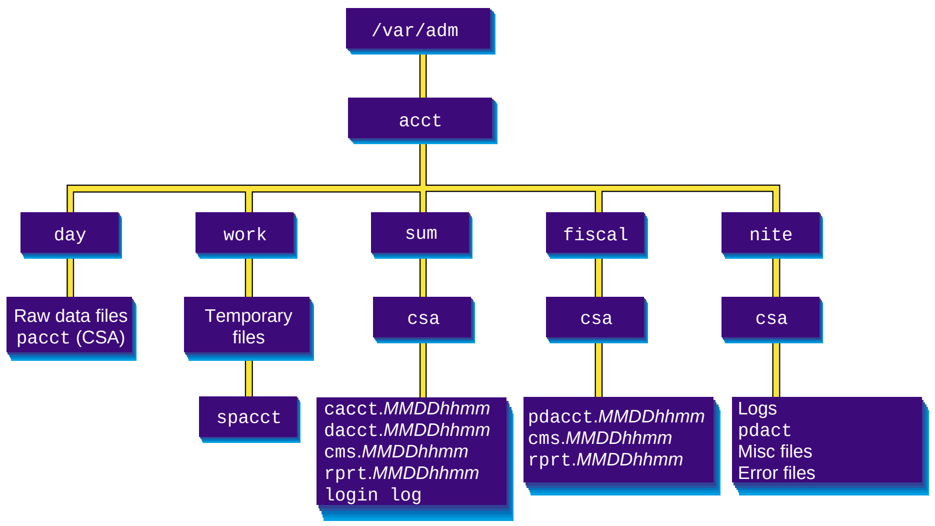

Figure 5-1 The `/var/adm/acct` Directory

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

Warning: On a IRIX security-enhanced system, the `csacom`(1) command is considered to be a covert channel. You may want to consider restricting access to this command to the `adm` group.

Files in the `/var/adm/acct/` Directory

The `/var/adm/acct` 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/adm/acct/day` at the start of an CSA daily accounting run and the `spacct` file.

`sum/csa`	Contains the cumulative daily accounting summary files and reports created by `csarun`(1M). The ASCII format is in `/var/adm/acct/sum/csa/rprt.`*MMDDhhmm*. The binary data is in `/var/adm/acct/sum/csa/cacct.`*MMDDhhmm*, `/var/adm/acct/sum/csa/cms.`*MMDDhhmm*, and `/var/adm/acct/sum/csa/dacct.`*MMDDhhmm*.
`fiscal/csa`	Contains periodic accounting summary files and reports created by `csaperiod`(1M). The ASCII format is in `/var/adm/acct/fiscal/csa/rprt.`*MMDDhhmm*. The binary data is in `/usr/adm/acct/fiscal/csa/cms.`*MMDDhhmm* and `/usr/adm/acct/fiscal/csa/pdacct.`*MMDDhhmm*.
`nite/csa`	Contains log files, `csarun` state, and execution times files.

Files in the `/var/adm/acct/day` Directory

The following files are located in the `/var/adm/acct/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/adm/acct/work` Directory

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

File	**Description**
`BAD.Wpacct*`	Unprocessed accounting data containing invalid records (verified by `csaverify`(1M)).
`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`(1M) (See the `dodisk`(1M) man page).
`Wdtmp`	Disk accounting data (ASCII) created by `dodisk`(1M).
`Wpacct*`	Raw process and daemon accounting data.
`spacct`	`sorted pacct` file.

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

The following data files are located in the `/var/adm/acct/sum/csa` 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/adm/acct/fiscal/csa` Directory

The following files are located in the `/var/adm/acct/fiscal/csa` 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/adm/acct/nite/csa` Directory

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

File	**Description**
`active`	Used by the `csarun`(1M) 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 86).
`diskcacct`	Disk accounting records in `cacct.h` format, created by `dodisk`.
`Eaddc`*MMDDhhmm*	Error/warning messages from the `csaaddc`(1M) command for an accounting run done on *MMDD* at *hhmm*.
`Earc1`*MMDDhhmm*	Error/warning messages from the `csa.archive1`(1M) command for an accounting run done on *MMDD* at *hhmm*.
`Earc2`*MMDDhhmm*	Error/warning messages from the `csa.archive2`(1M) command for an accounting run done on *MMDD* at *hhmm*.
`Ebld.`*MMDDhhmm*	Error/warning messages from the `csabuild`(1M) command for an accounting run done on *MMDD* at *hhmm*.
`Ecmd.`*MMDDhhmm*	Error/warning messages from the `csacms`(1M) command when generating an ASCII report for an accounting run done on *MMDD* at *hhmm*.
`Ecms.`*MMDDhhmm*	Error/warning messages from the `csacms`(1M) command when generating binary data for an accounting run done on *MMDD* at *hhmm*.

`Econ.`*MMDDhhmm*	Error/warning messages from the `csacon`(1M) command for an accounting run done on *MMDD* at *hhmm*.
`Ecrep.`*MMDDhhmm*	Error/warning messages from the `csacrep`(1M) command for an accounting run done on *MMDD* at *hhmm*.
`Ecrpt.`*MMDDhhmm*	Error/warning messages from the `csacrep`(1M) command for an accounting run done on *MMDD* at *hhmm*.
`Edrpt.`*MMDDhhmm*	Error/warning messages from the `csadrep`(1M) command for an accounting run done on *MMDD* at *hhmm*.
`Erec.`*MMDDhhmm*	Error/warning messages from the `csarecy`(1M) command for an accounting run done on *MMDD* at *hhmm*.
`Euser.`*MMDDhhmm*	Error/warning messages from the `csa.user`(1M) user exit for an accounting run done on *MMDD* at *hhmm*.
`Epuser.`*MMDDhhmm*	Error/warning messages from the `csa.puser`(1M) user exit for an accounting run done on *MMDD* at *hhmm*.
`Ever.tmp1`*MMDDhhmm*	Output file from invalid record offsets from the `csaverify`(1M) command for an accounting run done on *MMDD* at *hhmm*.
`Ever.tmp2`*MMDDhhmm*	Error/warning messages from the `csaverify`(1M) command for an accounting run done on *MMDD* at *hhmm*.
`Ever.`*MMDDhhmm*	Error/warning messages from the `csaedit`(1M) and `csaverify`(1M) 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 86).
`lock lock1`	Used to control serial use of the `csarun`(1M) comand.
`pd2log`	Diagnostic output created during execution of `csaperiod` (see `cron` entry for `csaperiod` in "Setting Up CSA", page 86).

`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/lib/acct` Directory

The `/usr/lib/acct` directory contains the following commands and shell scripts used by CSA:

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 user commands associated with CSA:

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

The `/usr/lib/acct` directory may also contain the following scripts if your site uses the accounting user exits:

Script	**Description**
`csa.archive1`	Site-generated user exit for `csarun`.
`csa.archive2`	Site-generated user exit for `csarun`.
`csa.fef`	Site-generated user exit for `csarun`.
`csa.user`	Site-generated user exit for `csarun`.
`csa.puser`	Site-generated user exit for `csaperiod`.

`/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/config` Directory

The `/etc/config` directory is the location of the `csaacct` file used by the `chkconfig`(1M) command. The `csaacct.options` contains options passed to the `csaswitch`(1M) command. Use a text editor to add any `csaswitch`(1M) options to be passed to `csaswitch` during system startup only.

Comprehensive System Accounting Expanded Description

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

- "Daily Operation Overview", page 85
- "Setting Up CSA", page 86
- "The `csarun` Command", page 90

Daily Operation Overview

When the IRIX 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/lib/acct/csaswitch` (see the `csaswitch`(1M) man page) command is called by `/etc/rc2`.

2. By default, `csa`, memory, and I/O record types are enabled in `/etc/csa.conf`. However, to run NQS, workload management, or tape daemon accounting you must modify the `/etc/csa.conf` file and the appropriate subsystem. For more information, see "Setting Up CSA", page 86.

3. The amount of disk space used by each user is determined periodically. The `/usr/lib/acct/dodisk` command (see `dodisk`(1M)) 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/lib/acct/csarun` is run (see `csarun`(1M)). 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/lib/acct/csachargefee` (see `csachargefee`(1M)). Each accounting period's fee file (`/var/adm/acct/day/fee`) is merged into the consolidated accounting records by `/usr/lib/acct/csaperiod` (see `csaperiod`(1M)).

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/lib/acct/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(1M) 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 104. As described in this section, CSA is run by a person with superuser permissions. CSA also can be run by users who are in the adm group and have the CAP_ACCT_MGT capability. See the capability(4) and capabilities(4) man pages for more information on the capability mechanism that provides fine grained control over the privileges of a process. See "Allowing Non Superusers to Execute CSA", page 117, for the necessary modifications.

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. Set NQS_START to on to enable NQS accounting. Set WKMG_START to on to enable workload management accounting. Set TAPE_START to on to enable tape accounting.

   b. If necessary, enable accounting from the daemon's side. Specifically, NQS, workload management, and tape accounting must also be enabled by the associated daemon. Use the qmgr set accounting on command to turn on NQS accounting. To enable tape daemon accounting, execute tmdaemon with the -c option. For more information on the tmdaemon command, see the *TMF Administrator's Guide*. To enable the workload management accounting, see the appropriate workload management guide for your system.

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`(1M) after updating the file. The `crontab` command automatically updates the `crontab` file and signals `cron`(1M) 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 /etc/chkconfig csaacct; then /usr/lib/acct/csarun 2> /var/adm/acct/nite/csa/fd2log; fi
0 2 *  * 4    if /etc/chkconfig csaacct; then /usr/lib/acct/dodisk -c > /var/adm/acct/nite/csa/dk2log; fi
5 * *  * 1-6  if /etc/chkconfig csaacct; then /usr/lib/acct/csackpacct; fi
0 5 1  * *    if /etc/chkconfig csaacct; then /usr/lib/acct/csaperiod -r  \
2> /var/adm/acct/nite/csa/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/crontabs/root so that `cron`(1M) automatically runs daily accounting:

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

The `csarun`(1m) 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.

The `dodisk` command must be invoked with the `-c` option. For more information, see the `dodisk`(1M) man page.

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

```
5 * *  * 1-6  if /etc/chkconfig csaacct; then /usr/lib/acct/csackpacct; fi
```

The `cron` command should periodically execute the `csackpacct`(1m) shell script. If the `pacct` file grows larger than 4000 1K blocks (default), `csackpacct` calls the command `/usr/lib/acct/csaswitch -c switch` to start a new `pacct` file. The `csackpacct` command also makes sure that there are at least 2000 1K blocks free on the file system containing /var/adm/acct (located in the /var directory by default). 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/adm/acct; the default directory is /var. 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 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/crontabs/root. This command generates a monthly report on all consolidated data files found in /var/adm/acct/sum/csa/* and then deletes those data files:

```
0 5 1  * *    if /etc/chkconfig csaacct; then /usr/lib/acct/csaperiod -r \
2> /var/adm/acct/nite/csa/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. On Trusted IRIX systems, perform the following steps::

   a. Ensure that user adm has the CAP_ACCT_MGT capability.

   b. Ensure that the following user exits (if they exist) are both readable and executable by user adm:

- /usr/lib/acct/csa.archive1
- /usr/lib/acct/csa.archive2
- /usr/lib/acct/csa.fef
- /usr/lib/acct/csa.puser

c. Include an entry similar to the one shown below in /var/spool/cron/crontabs/root:

```
 2 * * 4 suattr -M dbadmin -C CAP_DAC_READ_SEARCH,CAP_DAC_WRITE,
CAP_FOWNER,CAP_MAC_READ+eip -c "if /etc/chkconfig csaacct;
then /usr/lib/acct/dodisk -c 2> /var/adm/acct/nite/csa/dk2log; fi"
```

d. Include entries similar to the ones shown below in /var/spool/cron/crontabs/adm:

```
0 4 * * 1-6 su adm -C CAP_ACCT_MGT+pi -c "if /etc/chkconfig csaacct;
then /usr/lib/acct/csarun 2> /var/adm/acct/nite/csa/fd2log; fi"
5 * * * 1-6 su adm -C CAP_ACCT_MGT+pi -c "if /etc/chkconfig csaacct;
then /usr/lib/acct/csackpacct; fi"
0 5 1 * * if /etc/chkconfig csaacct;
then /usr/lib/acct/csaperiod -r 2> /var/adm/acct/nite/csa/pd2log; fi
```

6. Update the holidays file. The file /usr/lib/acct/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. The format is composed of three types of entries:

   - Comment Lines, which may appear anywhere in the file as long as the first character in the line is an asterisk.
   - Year Designation Line, which should be the first data line (noncomment line) in the file and must appear only once. The line consists of three fields of four digits each (leading white space is ignored). For example, to specify the year as 1992, prime time at 9:00 a.m., and nonprime time at 4:30 p.m., the following entry is appropriate:

     ```
     1992 0900 1630
     ```

     A special condition allowed for in the time field is that the time 2400 is automatically converted to 0000

- Company Holidays Lines, which 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 a number in the range of 1 through 366, indicating the day for the corresponding holiday (leading white space is ignored). The other three fields are actually commentary and are not currently used by other programs.

The `csarun` Command

The `/usr/lib/acct/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/adm/acct/nite/csa/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/adm/acct/nite/csa`. Both files should specify that the last invocation completed successfully. See "Restarting `csarun`", page 93.

Error and Status Messages

The `csarun` error and status messages are placed in the `/var/adm/acct/nite/csa` 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 `SYSLOG` 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 reentrant states so that `csarun` can be restarted. As each state completes, `/var/adm/acct/nite/csa/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/adm/acct/work/`*MMDD/hhmm* directory. File names are prefaced with `W`. `/var/adm/acct/nite/csa/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/adm/acct/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/lib/acct/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/lib/acct/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/adm/acct/sum/csa/cms.`*MMDDhhmm* for use by `csaperiod`.
`REPORT`	Generates the daily accounting report and puts it into `/var/adm/acct/sum/csa/rprt.`*MMDDhhmm*. A consolidated data file, `/var/adm/acct/sum/csa/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/adm/acct/sum/csa/rprt.`*MMDDhhmm*.
`FEF`	Third user exit of the `csarun` script. If a script named `/var/lib/acct/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/lib/acct/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/adm/acct/nite/csa/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/adm/acct/nite/csa/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/adm/acct/nite/csa/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/adm/acct/nite/csa/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/adm/acct/work/MMDD/hhmm
/var/adm/acct/sum/csa
```

If you are restarting at state `ARCHIVE2`, `CMS`, `REPORT`, `DREP`, or `FEF`, the `sorted pacct` file must be in `/var/adm/acct/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`(1M) 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`(1M) 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/lib/acct/csaverify -P pacct -o offsetfile
acct.cat-330 /usr/lib/acct/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/lib/acct/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/lib/acct/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 5-2.

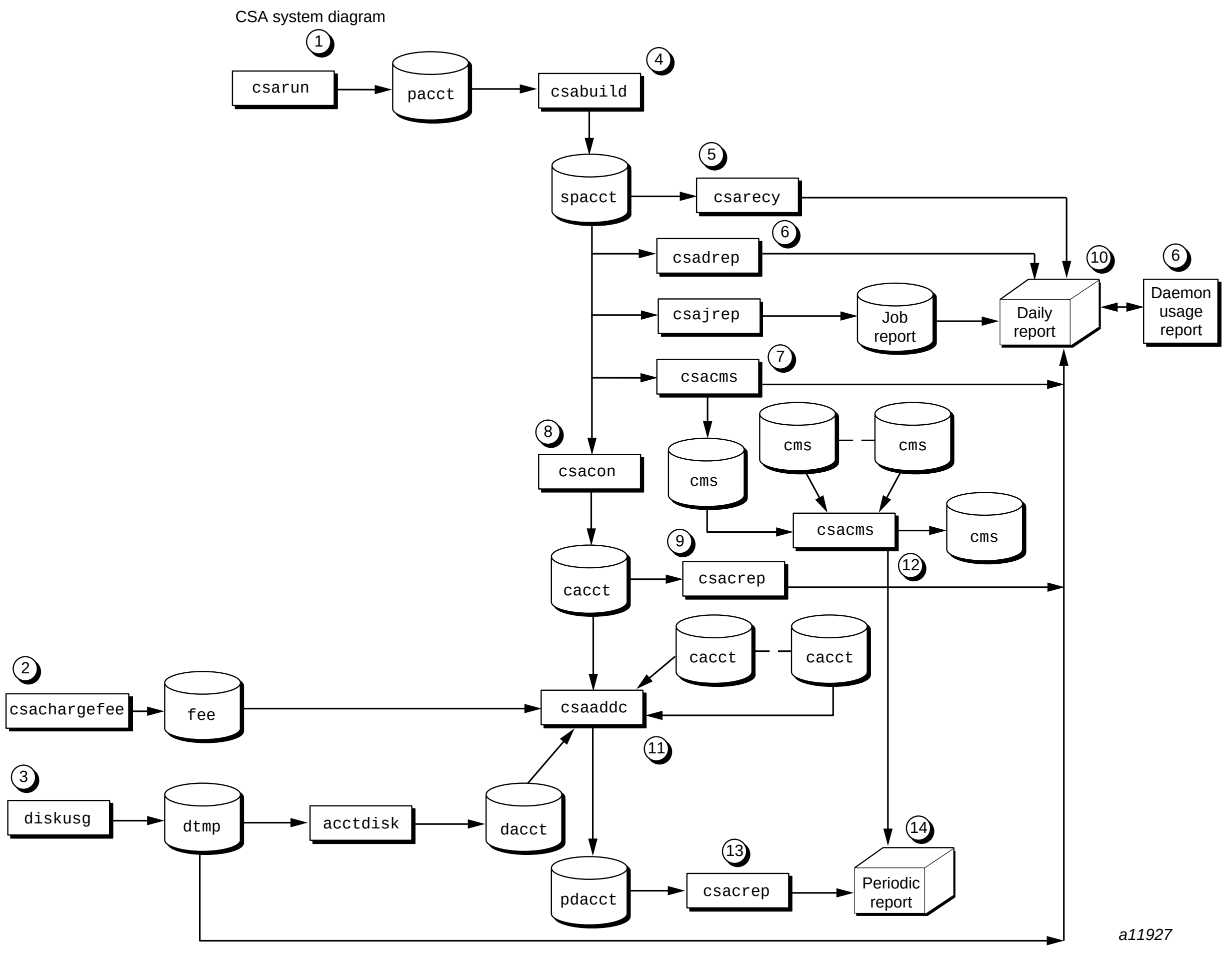

Figure 5-2 CSA Data Processing

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`(1m) command. The `csachargefee` command creates a fee file that is processed by `csaaddc`(1m).

3. Produce disk usage statistics. The `dodisk`(1m) 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`(1M) 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`(1M) 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`(1M) 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 98).

6. Generate the daemon usage report, which is appended to the daily report. `csadrep`(1m) reports usage of the NQS, workload management, and tape daemons. Input is either from a `sorted pacct` file created by `csabuild`(1M) 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`(1m) 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/adm/acct/sum/csa/cms.`*MMDDhhmm*, using the `cms` format. The `csacms`(1m) command can also create an ASCII file.

8. Condense records from the `sorted pacct` file. The `csacon`(1M) command condenses records from the `sorted pacct` file and writes consolidated records in `cacct` format to `var/adm/acct/sum/csa/cacct.`*MMDDhhmm*.
9. Generate an accounting report based on the consolidated data. The `csacrep`(1m) command generates reports from data in `cacct` format, such as output from the `csacon`(1M) 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 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`(1M) 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`(1m) 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`(1m) 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`(1m). Periodic (monthly) accounting (steps 12 through 14) is initiated by the `csaperiod`(1m) command. Daily and periodic accounting, as well as fee and disk usage generation (steps 2 through 3), can be scheduled by `cron`(1m) to execute regularly. See "Setting Up CSA", page 86, 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`.

`csarun` puts recycled data in the `/var/adm/acct/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 NQS daemon or workload management daemon delivers an NQS or workload management request's output, the request terminates. The daemon then writes an `NQ_DISP` record type for NQS or `WM_TERM` record type for workload management to the `pacct` accounting file, while the kernel writes an `EOJ` record to the `pacct` file.

Unlike interactive jobs, NQS or 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 pipe clients (NQS only), net clients, and checkpointed portions of the request. The pipe client and net client perform NQS or workload management processing on behalf of the request. The Load Sharing Facility (LSF) system currently does not support net clients.

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 an NQS request, and there is both an `EOJ` record for the request and an `NQ_DISP` record type in the `pacct` file.
- The job is a workload management request, and there is both an `EOJ` record for the request and an `WM_TERM` 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.
- The job is manually terminated by the administrator using one of the methods described in "How to Remove Recycled Data", page 100.

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/adm/acct/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`(1m) command can turn off accounting when there is not enough space on the file system containing `/var/adm/acct/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 101. 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/lib/acct/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 NQS 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 NQS requests, execute the `qstat` command, as follows, using a login that has either NQS manager or NQS 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 NQS 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/lib/acct/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/adm/acct/day` directory.

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

  ```
  rm /var/adm/acct/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 NQS or workload management requests being flagged as interactive jobs, or NQS or workload management requests being billed at the wrong queue rate. This is explained in detail in "NQS or Workload Management Requests and Recycled Data", page 103.

- 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 100.

Table 5-1 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`.
`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.

NQS or Workload Management Requests and Recycled Data

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

- CSA fails to flag the job as an NQS or workload management job. This causes the request to be billed at standard rates instead of an NQS or workload management queue rate (see "NQS SBUs", page 108 or "Workload Management SBUs", page 109).
- The request is billed at the wrong queue rate.
- The wrong queue wait time is associated with the request.

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

NQS or workload management accounting records are only written under the following circumstances:

- The NQS or workload management daemon receives a request.
- A request is routed to a queue. (NQS only)
- A request executes. This includes executing a request for the first time, restarting, and rerunning a request.
- A request terminates. An NQS request can terminate because it is completed, requeued, preempted, held, or rerun. 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 NQS or 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 NQS or 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
- Writing a user exit
- Modifying the charging of NQS or workload management jobs based on NQS or workload management termination status
- Tailoring CSA shell scripts

- Using `at`(1) instead of `cron`(1m) 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/lib/acct/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 = (NQS queue SBU value) * (sum of all process record SBUs*
+ sum of all tape record SBUs)

or

*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/lib/acct/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_QTIME`	Run queue wait time weight factor. The unit used for this weight is *billing units* per second. `P_QTIME` is multiplied by the run queue wait 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_QTIME * qwtime +
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_QTIME * qwtime +
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

NQS SBUs

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

The `NQS_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 `NQS_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 `NQS_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-NQS 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 `NQS_NUM_MACHINES` parameter sets the number of originating machines for which you want to set SBUs (the value must be at least 1). Each `NQS_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

`NQS_NUM_MACHINES`). SBUs for originating machines not specified in `/etc/csa.conf` are automatically set to 1.0.

The queue and machine SBUs are multiplied together to give an NQS multiplier. If the SBUs are set to less than 1.0, there is an incentive to run jobs in these queues or from these machines. SBUs of 1.0 indicate that jobs in the queues or from associated hosts are billed normally.

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

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 following:

- Number of mounts
- Device reservation time (seconds)

- Number of bytes read
- Number of bytes written

Example SBU Settings

The following shows how you could set up the SBU system. This example is restricted to the process records.

All time is considered prime time. Therefore, the nonprime time SBUs should be set to the same values as their prime time counterparts.

Users are charged $10 per hour of user CPU time. This is equal to $10 per 3600 seconds, which is $0.002777777777777 per second (`P_UTIME`).

Therefore, the charges are as follows (the nonprime time SBUs are set to the same values as their prime time counterparts):

Weight Factor	Charge
`P_BASIC`	1.0
`P_TIME`	1.0
`P_STIME`	0.0
`P_UTIME`	0.002777777777777
`P_QTIME`	0.0
`P_BWTIME`	0.0
`P_RWTIME`	0.0
`P_MEM`	0.0
`P_XMEM`	0.0
`P_VMEM`	0.0
`P_IO`	0.0
`P_BIO`	0.0
`P_CIO`	0.0
`P_LIO`	0.0

Daemon Accounting

Accounting information is available from the NQS, workload management, and online tape daemons. Data is written to the `pacct` file in the `/var/adm/acct/day` directory.

In most cases, daemon accounting must be enabled by both the CSA subsystem and the daemon. "Setting Up CSA", page 86, 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/lib/acct/csaswitch -c on -n tape
```

The NQS or workload management, and online tape daemon, also, must enable accounting. Use the `qmgr set accounting on` command to turn on NQS accounting. Tape daemon accounting is enabled when `tmdaemon`(1m) is executed with the `-c` option.See the appropriate workload management guide for information on how to enable workload management accounting.

Note: If you are running the Load Sharing Facility (LSF) system and want to enable workload management accounting, you must set two LSF configuration variables in the `lsf.conf` file as follows:

```
LSF_ENABLE_CSA=y
LSF_ULDB_DOMAIN = <ULDB_domain_name>
```

If `LSF_ENABLE_CSA` is defined in the `lsf.conf` file, LSF writes LSF batch job events to the `pacct` file for processing through CSA. For LSF job accounting, records are written to `pacct` at the start and end of each LSF job.

If a ULDB domain for LSF is defined in the `lsf.conf` file, LSF creates an IRIX job and applies the configured resource limits to it. LSF resource limits defined in `lsb.queues` or at job submission override IRIX job limits defined in the ULDB.

For more information on the Load Sharing Facility (LSF) system and workload management accounting, see the appropriate LSF documentation.

Daemon accounting is disabled at system shutdown (see "Setting Up CSA", page 86). It can also be disabled at any time by the `csaswitch` command when used with the `off` operand. For example, to disable NQS accounting, execute the following command:

```
/usr/lib/acct/csaswitch -c off -n nqs
```

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/lib/acct/csa.archive1`
`ARCHIVE2`	`/usr/lib/acct/csa.archive2`
`FEF`	`/var/lib/acct/csa.fef`
`USEREXIT`	`/usr/lib/acct/csa.user`

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

`csaperiod` **state**	**User exit**
`USEREXIT`	`/usr/lib/acct/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.

If CSA is run by a user without superuser permissions, the user exits must be both readable and executable by this user (see "Allowing Non Superusers to Execute CSA", page 117).

Writing a User Exit

This section provides information about writing a user exit. The first example shows a user exit that saves the `sorted pacct` file after a daily accounting run. The second example shows a user exit that consolidates information for a daily report by project rather than by user.

Example 5-1 Save a `sorted pacct` File During a Daily Accounting Run

The `csarun`(1M) and `csaperiod`(1M) scripts use shell variables that are available for use within a user exit script. For example, the `sorted pacct` file is deleted after a successful daily accounting run. However, if you want to save that file, you could use any of the user exits that are executed after the `sorted pacct` file is created (see the `csarun`(1M) man page). Here is a simple user exit script to do just that:

```
#! /bin/sh
echo "Copying spacct file to /tmp/spacct"
cp ${SPACCT} /tmp/spacct
```

Example 5-2 Consolidated Information Report by Project Rather than by User

The default output for consolidated information from a daily report is as follows:

```
CONSOLIDATED INFORMATION REPORT BETWEEN 08/09 04:00   AND   08/09 14:48

PROJECT    USER    LOGIN    CPU-TIM  KCORE *  KVIRT *   IOWAIT [SECS]
 NAME       ID     NAME     [SECS]   CPU-MIN  CPU-MIN   BLOCK     RAW
======== ======== ======== ======== ======== ======== ======== ========
sysadm   0        root           30      536     1177       48        0
root     4        sys             0        5       11        0        0
csa      5        adm             5       24      194        1        0
root     1461     security        1        2       16        0        0
nqe      10320    user12          2        5       68        1        0
```

To show consolidated information for a daily report by project rather than by user, use the `csacon`(1M) and `csacrep`(1M) commands with the project option as follows:

```
/usr/lib/acct/csacon -Ap -s /tmp/spacct > /tmp/cacct_p
/usr/lib/acct/csacrep -hpcw < /tmp/cacct_p > /tmp/csacrep.out.p
```

The output is as follows:

```
PROJECT    USER    LOGIN    CPU-TIM  KCORE *  KVIRT *   IOWAIT [SECS]
 NAME       ID     NAME     [SECS]   CPU-MIN  CPU-MIN   BLOCK     RAW
======== ======== ======== ======== ======== ======== ======== ========
root     Unknown  Unknown         1        8       28        0        0
sysadm   Unknown  Unknown        31      537     1187       49        0
csa      Unknown  Unknown         5       24      194        1        0
nqe      Unknown  Unknown         2        7       83        1        0
```

The example `/usr/lib/acct/csa.user` script below performs the same operation as the `csacon`(1M) and `csacrep`(1M) commands example above to include a consolidated information by project report within the daily report:

```
#!/sbin/sh
#
csacon ${ALLJOBS} -p -s ${SPACCT} > ${SUM_DIR}/cacct_p.${DTIME} \
        2> ${NITE_DIR}/Econ.${DTIME}
if [ ${?} -ne 0 ]
then
        CSAERRMSG="REPORT - csacon errors \
                \n\tSee ${NITE_DIR}/Econ.${DTIME} and/or ${NITE_DIR}/fd2log"
        ERROR_EXIT
fi
chgrp ${CHGRP} ${SUM_DIR}/cacct_p.${DTIME}
#
csacrep -hpcw < ${SUM_DIR}/cacct_p.${DTIME} \
> ${SUM_DIR}/conrpt_p.${DTIME} 2> ${NITE_DIR}/Ecrpt_p.${DTIME}
if [ ${?} -ne 0 ]
then
        CSAERRMSG="REPORT - csacrep errors \
                \n\tSee ${NITE_DIR}/Ecrep_p.${DTIME} and/or ${NITE_DIR}/fd2log"
        ERROR_EXIT
fi
#
cd ${SUM_DIR}
echo "${RPTHDR}\n" > tmprprt
echo "Put some header message here\n"  >> tmprprt
cat conrpt_p.${DTIME} >> tmprprt
pr -h "${DAYHDR} ${SYSNAME} ${RELMSG}" tmprprt >> rprt.${DTIME}
#
```

If you want the new binary data files (`cacct_p` in the user exit example, above) to be used with the periodic report, you need to create a user exit for `/usr/lib/acct/csaperiod`.

Charging for NQS Jobs

By default, SBUs are calculated for all NQS jobs regardless of the job's NQS termination code. If you do not want to bill portions of an NQS request, set the appropriate `NQS_TERM_`*xxxx* variable (termination code) in the `/etc/csa.conf` file to `0`, which 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**
`NQS_TERM_EXIT`	Generated when the request finishes running and is no longer in a queued state. At NQS shutdown time, requests that specified both the `-nc` (no checkpoint) and `-nr` (no rerun) options for `qsub` also have `NQS_TERM_EXIT` records written. In addition, this record is written for requests that specified the `-nr` option for `qsub` and were running at the time of a system crash.
`NQS_TERM_REQUEUE`	Written for running requests that are checkpointed and then requeued when NQS shuts down.
`NQS_TERM_PREEMPT`	Written when a request is preempted with the `qmgr preempt request` command.
`NQS_TERM_HOLD`	Written for a request that is checkpointed with the `qmgr hold request` command. The `hold request` command differs from the checkpoint done at daemon shutdown time because a "hold" keeps the job from being scheduled until a `qmgr release` command is executed.
`NQS_TERM_OPRERUN`	Written when a request is rerun with the `qmgr rerun request` command. At NQS shutdown time, jobs that cannot be checkpointed and do not have the `-nr` (no rerun) option

	for `qsub` specified have this type of termination record written. The requests are requeued with this status.
`NQS_TERM_RERUN`	Written when a request is a non-operator rerun request.

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. 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/adm/acct` 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/lib/acct/csa.user`, that is executed from the `USEREXIT` section of `csarun`. For more information, see "Setting up User Exits", page 112.

Allowing Non Superusers to Execute CSA

Your site may want to allow users without superuser permissions to run CSA accounting. CSA can be run by users who are in the group `adm` and have the `CAP_ACCT_MGT` capability. See the `capability`(4) and `capabilities`(4) man pages for more information on the capability mechanism that provides fine grained control over the privileges of a process.

The following steps describe the process of setting up CSA so it is executed automatically on a daily and periodic basis by a user without superuser permissions. In this example, the user without superuser permissions is `adm`:

1. Ensure that user `adm` is a member of group `adm` and has the `CAP_ACCT_MGT` capability.

2. Ensure that the following user exits (if they exist) are both readable and executable by user `adm`:

   - `/usr/lib/acct/csa.archive1`
   - `/usr/lib/acct/csa.archive2`
   - `/usr/lib/acct/csa.fef`

- `/usr/lib/acct/csa.user`
- `/usr/lib/acct/csa.puser`

3. Follow steps 1 through 5 of "Setting Up CSA", page 86, to set up system billing units, record system boot times, and turn off accounting before system shutdown.

4. Include an entry similar to the one shown below in `/var/spool/cron/crontabs/root` so that `cron` automatically runs `dodisk`(1m):

```
0 2 * * 4 if /etc/chkconfig csaacct; then /usr/lib/acct/dodisk -c 2> /var/adm/acct/nite/csa/dk2log; fi
```

   The `dodisk` command must be executed by `root`, because no other user has the correct permissions to read `/dev/dsk/*`. For more information on the `dodisk`(1M) command, see the `dodisk`(1M) man page.

5. Include entries similar to the ones shown below in `/var/spool/cron/crontabs/adm` so that user `adm` automatically runs daily accounting by using `cron`:

```
0 4 * * 1-6 su adm -C CAP_ACCT_MGT+pi -c "if /etc/chkconfig csaacct;
then /usr/lib/acct/csarun 2> /var/adm/acct/nite/csa/fd2log; fi"
5 * * * 1-6 su adm -C CAP_ACCT_MGT+pi -c "if /etc/chkconfig csaacct;
then /usr/lib/acct/csackpacct; fi"
```

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

6. To run monthly accounting, place an entry similar to the one below in `/var/spool/cron/crontabs/adm` (this command generates a monthly report on all consolidated data files found in `/var/adm/acct/sum/csa` and then deletes those data files):

```
Change the crontab entry for #6 to the following:
  0 5 1 * * if /etc/chkconfig csaacct;
then /usr/lib/acct/csaperiod -r 2> /var/adm/acct/nite/csa/pd2log; fi
```

7. Update the holidays file as described in "Setting Up CSA", page 86.

Note: The `cron` entries listed above only work when the login shell of user `adm` is `sh` or `ksh`.

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/lib/acct/csarun 2> /var/adm/acct/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/adm/acct/sum/csa` directory; periodic reports are located in the `/var/adm/acct/fiscal/csa` 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.

CSA Daily Report

This section describes the following reports:

Consolidated Information Report

The Consolidated Information Report is sorted by user ID and then project ID. 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
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
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 NQS or workload management, and tape daemons. This report has several individual reports depending upon if there was NQS, workload management, or tape daemon activity within this reporting period.

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

Heading	**Description**
Job Type	Type of job (interactive or NQS 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 NQS or workload management jobs

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

Heading	**Description**
Job Type	Type of job (interactive or NQS 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 Tape Usage Report gives the NQS or workload management and interactive job usage related to tape activity for these jobs.

Heading	**Description**
Job Type	Type of job (interactive or NQS or workload management)
Device Group	Tape device group name
Rsv Time	Tape reservation time in seconds
Mounts	Number of tape mounts
KBytes Read	Tape amount read in Kbytes
KBytes Written	Tape amount written in Kbytes
User CPU	Amount of user CPU time used in seconds
Sys CPU	Amount of system CPU time used in seconds

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

Queue Name	Name of the NQS or 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 124
- "Command summary report", page 124

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`(1M)
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 and Existing IRIX Software

This section describes some changes and additions to existing documentation for the IRIX operating system.

`acct`(1M) Man Page

The `acctdisk` command contains a `-c` option that reads standard input and converts records to `cacct` format, which it writes to standard output.

`acctsh`(1M) Man Page

The `lastlogin`(1M) command contains a `-c` option with an *infile* argument that specifies that `lastlogin` should process *infile*, which is a consolidated accounting file in `cacct` format.

The `dodisk` command information is now contained in a new `dodisk`(1M) man page.

`dodisk`(1M) Man Page

The IRIX 6.5.8 release introduced a new `dodisk`(1M) man page. The `dodisk` command information was previously in the `acctsh`(1M) man page.

`explain`(1) Man Page

CSA uses the message catalog system. There are two files that CSA uses for the message catalog:

- `/usr/lib/locale/C/LC_MESSAGES/acct.cat`
- `/usr/lib/locale/C/LC_MESSAGES/acct.exp`

The group code `acct` for the CSA Software Product has been added to the `explain`(1) page in the 6.5.8f release of the IRIX operating system.

`capabilities`(4) Man Page

Basic accounting and CSA require the same capability. `CAP_ACCT_MGT` is the privilege required to use accounting setup system calls, `acct`(2). The same privilege is required to use the new `acctctl`(3c) call. `acctctl`(3c) has been added to the `capabilities`(4) man page in the 6.5.8f release of the IRIX operating system.

Migrating Accounting Data

No changes have been made to basic accounting or extended accounting records. There is no migration of accounting data between these two IRIX accounting methods and CSA. That is, basic accounting commands should continue to be used with basic accounting, and third party packages should continue to be used with extended accounting data.

CSA accounting commands can only be used with CSA accounting data. CSA commands cannot process basic accounting or extended accounting records. Basic accounting commands cannot process CSA generated accounting data.

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 pages are provided with CSA software:

Administrator man page	**Description**
`csaaddc`(1m)	Combines `cacct` records.
`csabuild`(1m)	Organizes accounting records into job records.
`csachargefee`(1m)	Charges a fee to a user.
`csackpacct`(1m)	Checks the size of the CSA process accounting file.
`csacms`(1m)	Summarizes command usage from per-process accounting records.
`csacon`(1m)	Condenses records from the `sorted pacct` file.
`csacrep`(1m)	Reports on consolidated accounting data.
`csadrep`(1m)	Reports daemon usage.
`csaedit`(1m)	Displays and edits the accounting information.

`csagetconfig`(1m)	Searches the accounting configuration file for the specified argument.
`csajrep`(1m)	Prints a job report from the `sorted pacct` file.
`csarecy`(1m)	Recycles unfinished jobs into the next accounting run.
`csaswitch`(1m)	Checks the status of, enables or disables the different types of CSA, and switches accounting files for maintainability.
`csaverify`(1m)	Verifies that the accounting records are valid.

Chapter 6

IRIX Memory Usage

This section describes commands that provide information about physical and virtual memory usage on the IRIX operating system.

This chapter contains the following sections:

- "Memory Usage Commands", page 129
- "Shared Memory", page 131
- "Physical Memory", page 132
- "Virtual Memory", page 132

Memory Usage Commands

Most of the memory usage commands provide a snapshot view of the current memory usage either on a per process basis or a per job basis.

Examples of per process commands are as follows:

- `gmemusage`(1)
- `pmem`(1)
- `top`(1)
- `ps`(1)

For more information on these commands, see the appropriate man page.

Per job commands include the following:

- `jstat`(1)

The Comprehensive System Accounting (CSA) commands, such as, `csacom`(1) and `ja`(1), provide historical memory usage information after a process or job terminates.

The `jstat`(1) command reports the current usage and highwater memory values of all concurrently running processes within a job.

If the `-l` option is specified, the `jstat` command will print out the current usage, high usage, current limit, and maximum limit information for the current job. (Note that `vmemory` is virtual memory and `ressetsize` is resident set size).

The following example shows the output of the `jstat -l` option:

```
 % jstat -l


         JID                OWNER          COMMAND
         ------------------ -------------- --------------
         0x106f             user1          -tcsh



         LIMIT NAME         USAGE          HIGH USAGE     CURRENT LIMIT  MAX LIMIT
         ------------------ -------------- -------------- -------------- --------------
         cputime            0              0              unlimited      unlimited
         datasize           272k           544k           unlimited      unlimited
         files              8              32             400            5000
         vmemory            4224k          14112k         unlimited      unlimited
         ressetsize         3520k          6384k          unlimited      unlimited
         threads            1              1              unlimited      unlimited
         processes          2              7              1024           1024
         physmem            3520k          6384k          unlimited      unlimited
```

The `-s` option of the `ja`(1) command reports the highwater memory value of the single largest process memory within a job.

It is not a cumulative highwater mark of all processes within the job since this value is gathered from the accounting records of terminated processes.

The following example shows the output of the `ja -s` option:

```
% ja -s

Job CSA Accounting - Summary Report
          ====================================

          Job Accounting File Name         : /tmp/ja.username
          Operating System                 : IRIX64 snow 6.5 10120733 IP27
          User Name (ID)                   : username (10320)
          Group Name (ID)                  : resmgmt (16061)
```

```
Project Name (ID)                 : CSA(40)
Array Session Handle              : 0x000000000000034b
Job ID                            : 0x310
Report Starts                     : 01/23/00 18:13:38
Report Ends                       : 01/23/00 18:17:05
Elapsed Time                      :         207      Seconds
User CPU Time                     :           0.9340 Seconds
System CPU Time                   :           0.0643 Seconds
Run Queue Wait Time               :           0.6463 Seconds
Block I/O Wait Time               :           0.1888 Seconds
Raw I/O Wait Time                 :           0.1323 Seconds
CPU Time Core Memory Integral     :           0.4305 Mbyte-seconds
CPU Time Virtual Memory Integral :           4.3298 Mbyte-seconds
Maximum Core Memory Used          :           0.1094 Mbytes
Maximum Virtual Memory Used       :          38.0000 Mbytes
Characters Read                   :           0.0603 Mbytes
Characters Written                :           0.0023 Mbytes
Blocks Read                       :           7
Blocks Written                    :           0
Logical I/O Read Requests         :          35
Logical I/O Write Requests        :          42
Number of Commands                :           7
System Billing Units              :           0.0000
```

The CSA memory integrals report the amount of memory used over CPU time, measured at clock intervals.

CSA, extended accounting, and the `jstat`(1) command all access the same kernel counters for per process memory size. Additional kernel counters accumulate these per process memory size values into job memory size values as reported by the `jstat` command. CSA does its accumulation into job values outside of the kernel.

Shared Memory

Both job limits and CSA report memory usage values for all processes in a job. Processes in the job can access shared memory segments. Those segments can be shared between processes in the job or with processes outside the job, depending on the type of shared memory segment involved. When determining the memory usage for the job as a whole, shared memory segments are counted once for each process that accesses the segment. This can result in a usage value that is much larger than expected. This is particularly true for parallel applications where a large number of processes share one or more memory segments.

Shared memory between processes is not prorated by CSA or the `jstat` command. The shared memory pages, both physical and virtual, are counted in the memory size for each process accessing the pages.

Physical Memory

The kernel calculates the physical highwater memory value, current usage value, and memory integral value at periodic intervals. These values are the resident set size for the process or job, but do not include pages associated with mapped devices (for example, a graphics device).

Virtual Memory

Unlike physical memory usage values, the kernel keeps virtual memory values continuously current in kernel counters. The kernel increments the CSA highwater value when the process virtual memory size increases. The `jstat` current usage and highwater value are set, as applicable, at periodic intervals in the kernel. The kernel also calculates the CSA virtual memory integral at periodic intervals.

These values include the virtual memory size (text, data, stack, shared memory, mapped files, shared libraries) for the process or job, but do not include pages associated with mapped devices (for example, a graphics device).

Programming Guide for Resource Management

This appendix contains information for job limits, the User Limits DataBase (ULDB), and cpusets system programming.

This appendix contains the following sections:

- "Application Programming Interface for Job Limits", page 133
- "Application Programming Interface for the ULDB", page 136
- "Application Programming Interface for the Cpuset System", page 140

Application Programming Interface for Job Limits

This section describes the data types and function calls used by the library interface to the application programming interface (API) functions.

Data Types

This section describes the specific data types that are used in the library interface to the API functions.

All limit values are specified by the `rlimit` structure defined for process limits in the `/usr/include/sys/resource.h` system include file:

```
typedef unsigned long rlim_t;
   struct rlimit_t {
           rlim_t        rlim_cur;
           rlim_t        rlim_max;
   };
```

The job ID is defined as a signed 64 bit value. It is treated opaquely by applications. The definition of `jid_t` resides in the `sys/types.h` system include file.

```
typedef int64_t jid_t;
```

Function Calls

The API for job limits is defined by a set of functions defined in the `libc.a` library. Each of the functions invokes the `syssgi`(2) system interface to perform the necessary operations. The function prototypes reside in the `/usr/include/sys/resource.h` system include file.

`getjlimit` and `setjlimit`

The `getjlimit` function retrieves limits on the consumption of a variety of system resources by a job and the `setjlimit` function sets these limits:

```
#include <sys/resource.h>
int getjlimit(jid_t jid, int resource, struct rlimit *rlp)
int setjlimit(jid_t jid, int resource, struct rlimit *rlp)
```

For additional information, see the `getjlimit`(2) man page.

`getjusage`

The `getjusage` function retrieves the resource usage values for the specified job ID:

```
#include <sys/resource.h>
int getjusage(jid_t jid, int resource, struct jobrusage *up)
```

If the *jid* parameter is zero, usage values for the current job will be returned. If *jid* is non-zero, it represents the job ID of the job for which usages values are retrieved. The *resource* parameter specifies the resource for which the usage values are returned. Allowable values are taken from the `JLIMIT_xxx` macros found in the `sys/resource.h` file. For example, the `JLIMIT_CPU` macro is for CPU time. The `up` parameter points to a `rusage` structure in the user program where the usage values will be returned.

For additional information, see the `getjusage`(2) man page.

`getjid`

The `getjid` function returns the job ID associated with the current process:

```
#include <sys/resource.h>
jid_t getjid(void);
```

For additional information, see the `getjid`(2) man page.

killjob

The `killjob` function sends a signal to all processes of the specified job ID:

```
#include <sys/resource.h>
int killjob(jid_t jid, int signal)
```

For additional information, see the `killjob`(2) man page.

jlimit_startjob

The `jlimit_startjob` function creates a new job and sets the job limits to the limit values in the ULDB.

The `jlimit_startjob` function follows:

```
#include <sys/resource.h>
jid_t    jlimit_startjob(char *username, uid_t uid, char *domainname);
```

For additional information, see the `jlimit_startjob`(2) man page.

makenewjob

The `makenewjob` function creates a new job container:

```
#include <sys/resource.h>
jid_t makenewjob(uid_t user, jid_t rjid)
```

For additional information, see the `makenewjob`(2) man page.

setwaitjobpid

The `setwaitjobpid` function sets a job to wait for a specified `pid` to call the `waitjob` function.

The `setwaitjobpid` function follows:

```
#include <sys/resource.h>
int setwaitjobpid(jid_t rjid, pid_t wpid)
```

For additional information, see the `setwaitjobpid`(2) man page.

waitjob

The `waitjob` function obtains information about a terminated job that has been set with `setwaitjobpid` argument to wait.

The `waitjob` function follows:

```
#include <sys/resource.h>
jid_t waitjob(job_info_t *jobinfo)
```

For additional information, see the `waitjob`(2) man page.

Error Messages

For error message information, see the appropriate man pages and "Error Messages", page 28.

Application Programming Interface for the ULDB

This section describes the data types and function calls used by the library interface to the ULDB.

Data Types

This section defines the specific data types that are used by the library interface to the user limits information. All ULDB definitions are in the `/usr/include/uldb.h` include file.

Binary limit values are held as unsigned 64 bit values as follows:

```
typedef rlim_t uldb_limit_t;
```

uldb_namelist_t

The `uldb_namelist_t` data type is used to contain name lists such as limit names, domain names, and so on. The `namelist` structure contains a count of the items and a pointer to a list of pointers to the names. The `uldb_namelist_t` data type is as follows:

```
typedef struct uldb_namelist_s {
      int uldb_nitems,                # number of names in the list
      char **uldb_names               # list of name pointers
   } uldb_namelist_t;
```

`uldb_limitlist_t`

The `uldb_limitlist_t` data type is used to contain a list of binary limit values. The limit list structure contains a count of the items and a pointer to an array of limit values. The `uldb_limitlist_t` data type follows:

```
typedef struct uldb_limitlist_s {
      int uldb_nitems,                # number of limit values in the list
      uldb_limit_t *uldb_limits       # list of limit pointers
   } uldb_limitlist_t;
```

Function Calls

This section defines the function calls that are used by the library interface to the user limits information.

The functions that retrieve limit values are as follows:

- `uldb_get_limit_values`
- `uldb_get_value_units`
- `uldb_get_limit_names`
- `uldb_get_domain_names`

`uldb_get_limit_values`

The `uldb_get_limit_values` function retrieves a set of limit values for a domain or user. If there is no explicit entry for the specified user, the domain defaults are returned. The set of limits requested is provided using the `uldb_namelist_t` structure. The returned limit list pointer references a new `uldb_limitlist_t` structure created by a call to the `malloc` routine that the application is responsible for freeing when the structure is no longer needed. The order of limit values in the returned `uldb_limitlist_t` structure corresponds to the order of limit names in the input `uldb_namelist_t` structure. If the user name is NULL, the list of limits for the domain are retrieved instead of the user limits.

An example of `uldb_get_limit_values` follows:

```
#include include/uldb.h
uldb_limitlist_t *        # returns pointer to limit list or NULL if error
   uldb_get_limit_values (         #
      char *domain_name,           # pointer to domain name
      char *user_name,             # name of user
      uldb_namelist_t *limits);    # namelist containing limit names
```

`uldb_get_value_units`

The `uldb_get_value_units` function returns a limit list structure containing the modifier values or units for the specified list of limits. The accepted modifier values are defined in the `uldb.h` header file. The returned list of names is provided by the `uldb_namelist_t` structure created by a call to the `malloc` function. The application is responsible for freeing this structure when it is no longer needed.

An example of `uldb_get_value_units` follows:

```
#include <include/uldb.h>
uldb_limitlist_t *        # returns pointer to limit list or NULL if error
   uldb_get_value_units (          #
      char *domain_name,           # pointer to domain name
      char *user_name,             # name of user
      uldb_namelist_t *limits);    # namelist containing limit names
```

`uldb_get_limit_names`

The `uldb_get_limit_names` function retrieves the complete list of limit names defined for a domain. The returned list of names is provided by the `uldb_namelist_t` structure created by a call to the `malloc` function. The application is responsible for freeing this structure when it is no longer needed.

An example of `uldb_get_limit_names` follows:

```
#include <include/uldb.h>
   uldb_namelist_t *               # returns pointer to name list or NULL if error
   uldb_get_limit_names (
       char *domain_name);         # pointer to domain name
```

uldb_get_domain_names

The `uldb_get_domain_names` function retrieves the complete list of domain names defined in the ULDB. The returned list of names is provided the `uldb_namelist_t` structure created by a call to the `malloc` function. The application is responsible for freeing this structure when it is no longer needed.

```
#include <include/uldb.h>
uldb_namelist_t *              # returns pointer to name list or NULL if error
   uldb_get_domain_names (
       void);
```

The functions that manage memory are as follows:

- `uldb_free_namelist`
- `uldb_free_limit_list`

uldb_free_namelist

The `uldb_free_namelist` function deletes a `namelist` structure and all its components.

An example of `uldb_free_namelist` follows:

```
#include <include/uldb.h>
void                                # returns 0 if okay, -1 on error
   uldb_free_namelist (               #
      uldb_namelist_t *names);      # pointer to namelist to be freed
```

uldb_free_limit_list

The `uldb_free_limit_list` function deletes a limitlist structure and all its components.

An example of `uldb_free_limit_list` follows:

```
#include <include/uldb.h>
void                                # returns 0 if okay, -1 on error
   uldb_free_limit_list (             #
      uldb_limit_list_t *limits);   # pointer to limit list to be freed
```

Error Messages

For error message information, see the `uldb_get_limit_values`(3c) and `jlimit_startjob`(3c) man pages or "Error Messages", page 28.

Application Programming Interface for the Cpuset System

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

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

The cpuset library is provided as a N32 Dynamic Shared Object (DSO) library. The library file is `libcpuset.so`, and it is normally located in the directory `/lib32`. Users of the library must include the `cpuset.h` header file which is located in `/usr/include`. The function interfaces provided in the cpuset library are declared as optional interfaces to allow for backwards compatibility as new interfaces are added to the library.

Note: The Cpuset library is only available on IRIX 6.5.8 and later releases.

It is possible to compile and run a program that uses this DSO and its interfaces if they are available, but continues to execute if they are missing. To do this, a replacement library for `libcpuset.so` must be made available. For an example of how to create a replacement library, see the `cpuset`(5) man page. For more information on DSO, see the `DSO`(5) man page.

The function interfaces within the cpuset library include:

Function interface	**Description**
`cpusetCreate`(3x)	Creates a cpuset

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

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

Management functions

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

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

cpusetCreate(3x)

NAME

`cpusetCreate` - creates a cpuset

SYNOPSIS

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

DESCRIPTION

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

The `qname` argument is the name that will be assigned to the new cpuset. The name of the cpuset must be a three to eight character string. Queue names having one or two characters are reserved for use by the IRIX operating system.

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

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

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

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

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

	requests cannot be satisfied, the allocating process will sleep until memory is available. The process will be killed if no more memory can be allocated. See policies below.
`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 (see `swap`(1M)) to free physical memory on the nodes contained in this cpuset. If swap space is exhausted, the process will be killed.
`CPUSET_POLICY_KIL`	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.

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

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

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

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

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

EXAMPLES

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

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

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

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

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

NOTES

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

SEE ALSO

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

DIAGNOSTICS

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

`ENODEV`	Request for CPU IDs that do not exist on the system.
`EPERM`	Request for CPU 0 as part of an exclusive cpuset is not permitted.

`cpusetAttach`(3x)

NAME

`cpusetAttach` - attaches the current process to a cpuset

SYNOPSIS

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

DESCRIPTION

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

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

EXAMPLES

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

```
char *qname = "mpi_set";

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

NOTES

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

SEE ALSO

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

DIAGNOSTICS

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

cpusetAttachPID(3x)

NAME

cpusetAttachPID - attach a specific process to a cpuset

SYNOPSIS

```
#include <cpuset.h>

     int cpusetAttachPID(qname, pid);

     char *qname;

     pid_t pid;
```

DESCRIPTION

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

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

EXAMPLES

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

```
char *qname = "mpi_set";

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

NOTES

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

SEE ALSO

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

DIAGNOSTICS

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

cpusetDetachPID(3x)

NAME

cpusetDetachPID - detach a specific process to a cpuset

SYNOPSIS

```
#include <cpuset.h>

     int cpusetDetachPID(qname, pid);

     char *qname;

     pid_t pid;
```

DESCRIPTION

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

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

EXAMPLES

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

```
char *qname = "mpi_set";

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

NOTES

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

SEE ALSO

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

DIAGNOSTICS

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

cpusetDetachAll(3x)

NAME

cpusetDetachAll - detaches all threads from a cpuset

SYNOPSIS

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

DESCRIPTION

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

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

EXAMPLES

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

```
char *qname = "mpi_set";

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

NOTES

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

SEE ALSO

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

DIAGNOSTICS

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

cpusetDestroy(3x)

NAME

`cpusetDestroy` - destroys a cpuset

SYNOPSIS

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

DESCRIPTION

The `cpusetDestroy` function is used to destroy the specified cpuset. The `qname` argument is the name of the cpuset that will be destroyed. Only processes running with root user ID are allowed to destroy cpuset queues. A cpuset can only be destroyed if there are no threads currently attached to it.

EXAMPLES

This example destroys the cpuset queue named `mpi_set`.

```
char *qname = "mpi_set";

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

NOTES

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

SEE ALSO

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

DIAGNOSTICS

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

Retrieval Functions

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

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

`cpusetGetCPUCount`(3x)

NAME

`cpusetGetCPUCount` - obtains the number of CPUs configured on the system

SYNOPSIS

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

DESCRIPTION

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

EXAMPLES

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

```
int ncpus;

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

NOTES

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

SEE ALSO

`cpuset`(1) and `cpuset`(5).

DIAGNOSTICS

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

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

cpusetGetCPUList(3x)

NAME

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

SYNOPSIS

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

DESCRIPTION

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

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

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

The `count` member is the number of CPU IDs in the list. The `list` member references the memory array that holds the list of CPU IDs. The memory for `list` is allocated when the `cpuset_CPUList_t` structure is allocated and it is released when the `cpuset_CPUList_t` structure is released.

EXAMPLES

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

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

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

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

        printf("CPUSET[%s] assigned CPUs:\n",
                qname);
```

NOTES

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

SEE ALSO

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

DIAGNOSTICS

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

cpusetGetName(3x)

NAME

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

SYNOPSIS

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

DESCRIPTION

The `cpusetGetName` function is used to obtain the name of the cpuset to which the specified process has been attached. The *pid* argument specifies the process ID. Currently, the only valid value for *pid* is 0, which returns the name of the cpuset to which the current process is attached.

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

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

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

The `list` member references the list of names.

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

`CPUSET_QUEUE_NAME`	Indicates that the corresponding name in `list` is the name of a cpuset queue

`CPUSET_CPU_NAME`	Indicates that the corresponding name in `list` is the CPU ID for a restricted CPU

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

EXAMPLES

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

```
cpuset_NameList_t *name;

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

NOTES

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

SEE ALSO

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

DIAGNOSTICS

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

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

`cpusetGetNameList`(3x)

NAME

`cpusetGetNameList` - gets the list of names for all defined cpusets

SYNOPSIS

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

DESCRIPTION

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

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

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

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

The `list` member references the list of names.

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

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

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

EXAMPLES

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

```
cpuset_NameList_t *names;

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

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

NOTES

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

SEE ALSO

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

DIAGNOSTICS

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

`cpusetGetPIDList`(3x)

NAME

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

SYNOPSIS

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

DESCRIPTION

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

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

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

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

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

EXAMPLES

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

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

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

NOTES

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

SEE ALSO

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

DIAGNOSTICS

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

cpusetGetProperties(3x)

NAME

`cpusetGetProperties` - retrieves various properties associated with a cpuset

SYNOPSIS

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

DESCRIPTION

The `cpusetGetProperties` function is used retrieve various properties identified by `qname` and returns a pointer to a `cpuset_Properties_t` structure. Every cpuset queue has a file that defines access permissions to the queue. The read permissions for that file will determine if a process owned by a specific user can retrieve the properties from the cpuset.

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

EXAMPLES

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

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


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

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

NOTES

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

SEE ALSO

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

DIAGNOSTICS

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

`cpusetAllocQueueDef`(3x)

NAME

`cpusetAllocQueueDef` - allocates a `cpuset_QueueDef_t` structure

SYNOPSIS

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

DESCRIPTION

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

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

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

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

`CPUSET_CPU_EXCLUSIVE`	Defines a cpuset to be restricted. Only threads attached to the cpuset queue (descendents of an attached thread inherit the attachement) may execute on the CPUs contained in the cpuset.
`CPUSET_MEMORY_LOCAL`	Threads assigned to the cpuset will attempt to assign memory only from nodes within the cpuset. Assignment of memory from outside the cpuset will occur only if no free memory is available from within the cpuset. No restrictions

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

`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 (see `swap`(1M)) to free physical memory on the nodes contained in this cpuset. If swap space is exhausted, the process will be killed.
`CPUSET_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.

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

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

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

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

The list member is the pointer to the list (an allocated array) of the CPU IDs. The memory for the list array is allocated and released when the cpuset_CPUList_t structure is allocated and released. The size of the list is determined by the count argument passed into the function cpusetAllocQueueDef.

EXAMPLES

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

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

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

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

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

NOTES

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

SEE ALSO

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

DIAGNOSTICS

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

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

Clean-up Functions

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

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

cpusetFreeQueueDef(3x)

NAME

cpusetFreeQueueDef - releases memory used by a cpuset_QueueDef_t structure

SYNOPSIS

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

DESCRIPTION

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

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

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

NOTES

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

SEE ALSO

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

`cpusetFreeCPUList`(3x)

NAME

`cpusetFreeCPUList` - releases memory used by a `cpuset_CPUList_t` structure

SYNOPSIS

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

DESCRIPTION

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

The `cpu` argument is the pointer to the `cpuset_CPUList_t` structure that will have it's memory released.

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

NOTES

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

SEE ALSO

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

`cpusetFreeNameList`(3x)

NAME

`cpusetFreeNameList` - releases memory used by a `cpuset_NameList_t` structure

SYNOPSIS

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

DESCRIPTION

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

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

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

NOTES

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

SEE ALSO

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

`cpusetFreePIDList`(3x)

NAME

`cpusetFreePIDList` - releases memory used by a `cpuset_PIDList_t` structure

SYNOPSIS

```
#include <cpuset.h>

 void cpusetFreePIDList(cpuset_PIDList_t *pid);
```

DESCRIPTION

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

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

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

NOTES

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

SEE ALSO

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

cpusetFreeProperties(3x)

NAME

cpusetFreeProperties - releases memory used by a cpuset_Properties_t structure

SYNOPSIS

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

DESCRIPTION

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

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

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

NOTES

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

SEE ALSO

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

Using the Cpuset Library

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

Example 6-1 Example of Creating a Cpuset

This example creates a cpuset named `myqueue` containing CPUs 4, 8, and 12. The example uses the interfaces in the cpuset library, `/lib32/libcpuset.so`, if they are present. If the interfaces are not present, it attempts to use the `cpuset`(1) command to create the cpuset.

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

#define PERMFILE "/usr/tmp/permfile"

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

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

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

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

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

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

```
        }
        return 0;
    }
```

Example 6-2 Example of Creating a Replacement Library

This example shows how to create a replacement library for `/lib32/libcpuset.so` so that a program built to use the cpuset library interfaces will execute if the library is not present.

1. Create the `replace.c` file that contains the following line of code:

   ```
   static void cpusetNULL(void) { }
   ```

2. Compile the `replace.c` file:

   ```
   cc -mips3 -n32 -c replace.c
   ```

3. Place the `replace.o` object created in the previous step in a library:

   ```
   ar ccrl libcpuset.a replace.o
   ```

4. Convert the library into a DSO:

   ```
   ld -mips3 -n32 -quickstart_info -nostdlib        \
      -elf -shared -all -soname libcpuset.so       \
      -no_unresolved -quickstart_info -set_version \
      sgi1.0 libcpuset.a -o libcpuset.so
   ```

5. Install the DSO on the system:

   ```
   install -F /opt/lib32 -m 444 -src libcpuset.so \
                        libcpuset.so
   ```

The replacement library can be installed in a directory defined by the `LD_LIBRARYN32_PATH` environment variable (see `rld`(1)). If the replacement library must be installed in a directory that is in the default search path for shared libraries, it should be installed in `/opt/lib32`.

Index