Resource Administration for Linux

007–4315–001

CONTRIBUTORS

Written by Terry Schultz

Edited by Rick Thompson

Illustrated by Chris Wengelski

Production by Glen Traefald

Engineering contributions by Marlys Kohnke and Todd Wyman

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

Record of Revision

Version	Description
001	June 2001 Original publication.

Contents

Figures

Tables

About This Manual

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

This manual contains the following chapters

Related Publications

For a list of CSA man pages, see "CSA Man Pages", page 90.

For a list of Array Services man pages, see .

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 can be found on the back cover.)

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.

Array Services

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

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

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

The Array Services package comprises the following primary components:

array daemon	These daemon processes, one in each node, cooperate to allocate ASH values and maintain information about node configuration and the relation of process IDs to ASHes
array configuration database	One copy at each node, this file describes the Array configuration for use by array daemons and user programs.
`ainfo` command	Lets the user or administrator query the Array configuration database and information about ASH values and processes.
`array` command	Executes a specified command on one or more nodes. Commands are predefined by the administrator in the configuration database.
`arshell` command	Starts a command remotely on a different node using the current ASH value.
`aview` command	Displays a multiwindow, graphical display of each node's status.

`ibarray` library	Library of functions that allow user programs to call on the services of array daemons and the array configuration database.

The use of the `ainfo`, `array`, `arshell`, and `aview` commands is covered in "Using an Array", page 2. The use of the `libarray` library is covered in"Array Services Library", page 30 .

Using an Array

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

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

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

- "Using an Array System", page 3, summarizes what a user needs to know, and the main facilities a user has available.
- "Managing Local Processes", page 5, reviews the conventional tools for listing and controlling processes within one node.
- "Using Array Services Commands", page 6, describes the common concepts, options, and environment variables used by the Array Services commands.
- " Interrogating the Array", page 11, summarizes how to use Array Services commands to learn about the Array and its workload, with examples.
- "Managing Distributed Processes", page 14, summarizes how to use Array Services commands to list and control processes in multiple nodes.

Using an Array System

As an ordinary user of an Array system you are a UNIX user with the additional benefit of being able to run distributed sessions on multiple nodes of the Array. You access the Array from either

- A workstation
- An X-terminal
- An ASCII terminal

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

Finding Basic Usage Information

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

- The name of the Array.

  You use this *arrayname* in Array Services commands.

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

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

- The hostnames of the array nodes.

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

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

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

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

Logging In to an Array

Each node in an Array has an associated hostname and IP network address. Typically, you use an Array by logging in to one node directly, or by logging in remotely from another host (such as the Array console or a networked workstation). For example,

from a workstation on the same network, this command would log you in to the node named *hydra6* as follows:

`rlogin` *hydra6*

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

The system administrators of your Array may choose to disallow direct node logins in order to schedule array resources. If your site is configured to disallow direct node logins, your administrators will be able to tell you how you are expected to submit work to the array-perhaps through remote execution software or batch queueing facilities.

Invoking a Program

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

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

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

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

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

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

  For example, C and Fortran programs that use parallel processing directives test the `MP_SET_NUMTHREADS` variable.
- MPI and PVM message-passing programs may require support files to describe how many tasks to invoke on which nodes.

Some information sources on program invocation are listed Table 1-1, page 5.

Table 1-1 Information Sources for Invoking a Program standard

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

Managing Local Processes

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

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

Monitoring Processes and System Usage

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

```
ps -elfj
```

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

Scheduling and Killing Local Processes

You can start a process at a reduced priority, so that it interferes less with other processes, using the nice command. If you use the csh shell, specify `/usr/bin/nice` to avoid the built-in shell command `nice`. To start a whole shell at low priority, use a command like the one that follows:

```
/bin/nice /bin/sh
```

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

```
kill -KILL 13032
```

Summary of Process Management Commands

Table 1-2, page 6, summarizes information about local process management.

Table 1-2 Information Sources: Local Process Management standard

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

Using Array Services Commands

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

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

This topic introduces the terms, concepts, and command options that are common to all Array Services commands as shown in .

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

About Array Sessions

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

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

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

About Names of Arrays and Nodes

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

```
% hostname
tokyo
```

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

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

About Authentication Keys

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

Summary of Common Command Option

The commands of Array Services that follow have a consistent set of command options: `ainfo`(1), `array`(1), `arshell`(1), `aview`(1), and `newsess`(1). Table 1-3 is a summary of these options. Not all options are valid with all commands; and each command has unique options besides those shown. The default values of some options are set by environment variables listed in the next topic.

Table 1-3 Array Services Command Option Summary

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

Specifying a Single Node

The -l and -s options work together. The -l (letter ell for local) option restricts the scope of a command to the node where the command is executed. By default, that is the node where the command is entered. When -l is not used, the scope of a query

command is all nodes of the array. The -s (server, or node, name) option directs the command to be executed on a specified node of the array. These options work together in query commands as follows:

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

Common Environment Variables

The Array Services commands depend on environment variables to define default values for the less-common command options. These variables are summarized in Table 2-6.

Table 1-4 Array Services Environment Variables

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

Interrogating the Array

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

Learning Array Names

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

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

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

Learning Node Names

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

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

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

Learning Node Features

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

```
homegrown 181% ainfo -s tokyo -l nodeinfo
Node information for server on machine "tokyo"
MACHINE tokyo
    VERSION  1.2
    8 PROCESSOR BOARDS
        BOARD: TYPE 15   SPEED 190
            CPU:   TYPE 9   REVISION 2.4
            FPU:   TYPE 9   REVISION 0.0
...
    16 IP INTERFACES  HOSTNAME tokyo   HOSTID 0xc01a5035
        DEVICE  et0    NETWORK    150.166.39.0    ADDRESS   150.166.39.39  UP
        DEVICE atm0    NETWORK 255.255.255.255    ADDRESS         0.0.0.0  UP
        DEVICE atm1    NETWORK 255.255.255.255    ADDRESS         0.0.0.0  UP
...
    0 GRAPHICS INTERFACES
    MEMORY
        512 MB MAIN MEMORY
        INTERLEAVE 4
```

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

Learning User Names and Workload

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

Learning User Names

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

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

Learning Workload

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

```
homegrown 181% array uptime
   homegrown:  up 1 day,  7:40,  26 users,  load average: 7.21, 6.35, 4.72
    disarray:  up  2:53,  0 user,  load average: 0.00, 0.00, 0.00
    datarray:  up  5:34,  1 user,  load average: 0.00, 0.00, 0.00
       tokyo:  up 7 days,  9:11,  17 users,  load average: 0.15, 0.31, 0.29
homegrown 182% array -l -s tokyo uptime
       tokyo:  up 7 days,  9:11,  17 users,  load average: 0.12, 0.30, 0.28
```

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

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

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

Managing Distributed Processes

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

About Array Session Handles (ASH)

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

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

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

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

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

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

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

This feature has little use at present. There is no existing command that can change its ASH, so you cannot assign the new ASH to another command. It is possible to write a program that takes an ASH from a command-line option and uses the Array Services function `setash()` to change to that ASH (however such a program must be privileged). No such program is distributed with Array Services 3.5 (but see "Managing Array Service Handles", page 35).

Listing Processes and ASH Values

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

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

```
homegrown 182% array -l -s tokyo ps | fgrep wombat
0x261cffff0000054c        tokyo 19007   wombat    0:00 -csh
0x261cffff0000054a        tokyo 17940   wombat    0:00 csh -c (setenv...
0x261cffff0000054c        tokyo 18941   wombat    0:00 csh -c (setenv...
0x261cffff0000054a        tokyo 17957   wombat    0:44 xem -geometry 84x42
0x261cffff0000054a        tokyo 17938   wombat    0:00 rshd
0x261cffff0000054a        tokyo 18022   wombat    0:00 /bin/csh -i
0x261cffff0000054a        tokyo 17980   wombat    0:03 /usr/gnu/lib/ema...
0x261cffff0000054c        tokyo 18928   wombat    0:00 rshd
```

Controlling Processes

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

Using arshell

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

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

You can use `arshell` to start a collection of unrelated programs in multiple nodes under a single ASH; then you can use the commands described under "Managing Session Processes", page 17 to stop, resume, or kill them.

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

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

---

1. Create a nested shell with a new ASH using `newsess`. Note the ASH value.
2. Within the new shell, start one or more programs using `arshell`.
3. Exit the nested shell.

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

About the Distributed Example

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

The `array` command is designed to permit the simultaneous launch of programs on all nodes with a single command. However, array can only launch programs that have been configured into it, in the Array Services configuration file. (The creation and management of this file is discussed under "About Array Configuration", page 18.)

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

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

```
        options nowait
```

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

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

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

Managing Session Processes

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

```
homegrown 175% ainfo ash
Array session handle of process 8912: 0x1111ffff0000032d
homegrown 176% newsess
homegrown 175% ainfo ash
Array session handle of process 8941: 0x11110000308b2fa6
```

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

```
homegrown 176% array spin
```

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

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

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

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

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

About Array Configuration

The system administrator has to initialize the Array configuration database, a file that is used by the Array Services daemon in executing almost every `ainfo` and `array` command. For details about array configuration, see the man pages cited in Table 1-5.

Table 1-5 Information Sources: Array Configuration

Topic	Man Page
Array Services overview	`array_services`(5)
Array Services user commands	`ainfo`(1) , `array`(1)
Array Services daemon overview	`arrayd`(1m)

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

About the Uses of the Configuration File

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

The configuration files inform the daemon of the data needed by `ainfo` and `array`:

- The names of Array systems, including the current Array but also any other Arrays on which a user could run an Array Services command (reported by `ainfo`).
- The names and types of the nodes in each named Array, especially the hostnames that would be used in an Array Services command (reported by `ainfo`).
- The authentication keys, if any, that must be used with Array Services commands (required as `-Kl` and `-Kr` command options, see "Summary of Common Command Option", page 8).
- The commands that are valid with the `array` command.

About Configuration File Format and Contents

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

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

Local option	Options that modify the operation of the other entries or `arrayd`.

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

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

Loading Configuration Data

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

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

- One file can have different access permissions than another. Typically, `/usr/lib/array/arrayd.conf` is world-readable and contains the available `array` commands, while `/usr/lib/array/arrayd.auth` is readable only by root and contains authentication codes.
- One node can have different configuration data than another. For example, certain commands might be defined only in certain nodes; or only the nodes used for interactive logins might know the names of all other nodes.
- You can use NFS-mounted configuration files. You could put a small configuration file on each machine to define the Array and authentication keys, but you could have a larger file defining `array` commands that is NFS-mounted from one node.

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

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

to kill the daemon, and

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

to kill and restart it in one operation.

Note: The script path name is `/etc/rc.d/init.d/array` on Linux systems.

The Array Services daemon in any node knows only the information in the configuration files available in that node. This can be an advantage, in that you can limit the use of particular nodes; but it does require that you take pains to keep common information synchronized. (An automated way to do this is sketched under "Designing New Array Commands", page 28.)

About Substitution Syntax

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

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

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

Testing Configuration Changes

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

After making a change, you can test an individual configuration file for correct syntax by executing `arrayd` as a command with the `-c` and `-f` options. For example,

suppose you have just added a new command definition to `/usr/lib/array/arrayd.local`. You can check its syntax with the command

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

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

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

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

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

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

Configuring Arrays and Machines

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

Specifying Arrayname and Machine Names

A simple example of an ARRAY definition is a follows:

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

The arrayname `simple` is the value the user must specify in the `-a` option (see "Summary of Common Command Option", page 8). One arrayname should be specified in a DESTINATION ARRAY local option as the default array (reported by `ainfo dflt`). Local options are listed under "Configuring Local Options", page 28.

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

Specifying IP Addresses and Ports

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

```
array simple
        machine congo
            hostname congo.engr.hitech.com
            port 8820
        machine niger
            hostname niger.engr.hitech.com
        machine nile
            hostname "198.206.32.85"
```

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

Specifying Additional Attributes

Under both ARRAY and MACHINE you can insert "attributes," which are named string values. These attributes are not used by Array Services, but they are displayed by `ainfo` and can be returned to programs using the Array Services library ("Array Services Library", page 30). Some examples of attributes would be as follows:

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

Tip: You can write code that fetches any arrayname, machine name, or attribute string from any node in the array. See "Database Interrogation", page 34.

Configuring Authentication Codes

In Array Services 3.5 only one type of authentication is provided: a simple numeric key that can be required with any Array Services command. You can specify a single authentication code number for each node. The user must specify the code with any command entered at that node, or addressed to that node using the `-s` option (see "Summary of Common Command Option", page 8).

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

Configuring Array Commands

The user can invoke arbitrary system commands on single nodes using the `arshell` command (see "Using arshell", page 15). The user can also launch MPI and PVM programs that automatically distribute over multiple nodes. However, the only way to launch coordinated system programs on all nodes at once is to use the `array`(1) command. This command does not accept any system command; it only permits execution of commands that the administrator has configured into the Array Services database.

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

Operation of Array Commands

When a user invokes the `array` command, the subcommand and its arguments are processed by the destination node specified by `-s`. Unless the `-l` option was given, that daemon also distributes the subcommand and its arguments to all other array

nodes that it knows about (the destination node might be configured with only a subset of nodes). At each node, `arrayd` searches the configuration database for a COMMAND entry with the same name as the array subcommand.

For example, when the user enters

```
array -s tokyo uptime
```

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

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

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

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

Summary of Command Definition Syntax

Look at the basic set of commands distributed with Array Services 3.5 (`/usr/lib/array/arrayd.conf`). Each COMMAND entry is defined using the subentries shown in Table 1-6. (These are described in great detail in man page `arrayd.conf`(4).)

Table 1-6 Subentries of a COMMAND Definition

Keyword	Meaning of Following Values
COMMAND	The name of the command as the user gives it to `array`.
INVOKE	A system command to be executed on every node. The argument values can be literals, or arguments given by the user, or other substitution values.

Keyword	Meaning of Following Values
MERGE	A system command to be executed only on the distributing node, to gather the streams of output from all nodes and combine them into a single stream.
USER	The userid under which the INVOKE and MERGE commands run. Usually given as USER %USER, so as to run as the user who invoked `array`.
GROUP	The groupname under which the INVOKE and MERGE commands run. Usually given as GROUP %GROUP, so as to run in the group of the user who invoked `array` (see reference page `groups`(1)).
PROJECT	The project under which the INVOKE and MERGE commands run. Usually given as PROJECT %PROJECT, so as to run in the project of the user who invoked `array` (see reference page `projects`(5)).
OPTIONS	A variety of options to modify this command; see Table 1-8 .

The system commands called by INVOKE and MERGE must be specified as full pathnames, because `arrayd` has no defined execution path. As with a shell script, these system commands are often composed from a few literal values and many substitution strings. The substitutions that are supported (which are documented in detail in the `arrayd.conf`(4) man page) are summarized in Table 1-7.

Table 1-7 Substitutions Used in a COMMAND Definition

Substitution	Replacement Value
%1..%9; %ARG(*n*); %ALLARGS; %OPTARG(*n*)	Argument tokens from the user's subcommand. %OPTARG does not produce an error message if the specified argument is omitted.
%USER, %GROUP, %PROJECT	The effective userid, effective groupid, and project of the user who invoked `array`.
%REALUSER, %REALGROUP	The real userid and real groupid of the user who invoked `array`.

Substitution	Replacement Value
%ASH	The ASH under which the INVOKE or MERGE command is to run.
%PID(*ash*)	List of PID values for a specified ASH. %PID(%ASH) is a common use.
%ARRAY	The arrayname, either default or as given in the `-a` option.
%LOCAL	The hostname of the executing node.
%ORIGIN	The full domain name of the node where the `array` command ran and the output is to be viewed.
%OUTFILE	List of names of temporary files, each containing the output from one node's INVOKE command (valid only in the MERGE subentry).

The OPTIONS subentry permits a number of important modifications of the command execution; these are summarized in Table 1-8.

Table 1-8 Options of the COMMAND Definition

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

Configuring Local Options

The LOCAL entry specifies options to `arrayd` itself. The most important options are summarized in Table 1-9.

Table 1-9 Subentries of the LOCAL Entry

Subentry	Purpose
DIR	Pathname for the `arrayd` working directory, which is the initial, current working directory of INVOKE and MERGE commands. The default is `/usr/lib/array`.
DESTINATION ARRAY	Name of the default array, used when the user omits the -a option. When only one ARRAY entry is given, it is the default destination.
USER, GROUP, PROJECT	Default values for COMMAND execution when USER, GROUP, or PROJECT are omitted from the COMMAND definition.
HOSTNAME	Value returned in this node by %LOCAL. Default is the hostname.
PORT	Socket to be used by `arrayd`.

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

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

Designing New Array Commands

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

Typically, a new command will be defined with an INVOKE subentry that names a script written in `sh`, `csh`, or `perl` syntax. You use the substitution values to set up arguments to the script. You use the USER, GROUP, PROJECT, and OPTIONS subentries to establish the execution conditions of the script. For one example of a command definition using a simple script, see "About the Distributed Example", page 16.

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

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

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

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

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

The command definition would be as follows:

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

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

Array Services Library

Array Services consists of a configuration database, a daemon (`arrayd`) that runs in each node to provide services, and several user-level commands. The facilities of Array Services are also available to developers through the Array Services library, a set of functions through which you can interrogate the configuration database and call on the services of `arrayd`.

The commands of Array Services are covered in "Using Array Services Commands", page 6. The administration of Array Services is described in "About Array Configuration", page 18 and topics that follow it. These topics are useful background information for understanding the Array Services library.

Array Services Library Overview

The programming interface to Array Services is declared in the header file `/usr/include/arraysvcs.h`. The object code is located in `/usr/lib/libarray.so`, included in a program by specifying `-larray` during compilation. The library is distributed in o32, n32, and 64-bit versions on IRIX and IA-64 versions on Linux (not all need to be installed).

The library functions can be grouped into these categories:

- Functions to connect to Array Services daemons in the local or other nodes, and to get and set `arrayd` options.
- Functions to interrogate the Array Services configuration database, listing arrays, nodes, and attributes of arrays and nodes.
- Functions to allocate Array Session Handles (ASHs), to query active ASHs and to change the relationship between PIDs and ASHs.
- A function to execute a command as for the `array` command (see "Operation of Array Commands", page 24).
- A function to execute any arbitrary user command on an array node.

These functions are examined in following topics.

Data Structures

The Array Services functions work with a number of data structures that are declared in `arraysvcs.h`. In general, each data structure is allocated by one particular function, which returns a pointer to the structure as the function's result. Your code uses the returned structure, possibly passing it as an argument to other functions.

When your code is finished with a structure, it is expected to call a specific function that frees that type of structure. If your code does not free each structure, a memory leak results.

The data structures and their contents are summarized in Table 1-10.

Table 1-10 Array Services Data Structures

Structure	Contents	Freed By Function
asarray_t	Name and attributes of an Array.	`asfreearray()`
asarraylist_t	List of *asarray_t* structures.	`asfreearraylist()`
asashlist_t	List of ASH values.	`asfreeashlist()`
ascmdrslt_t	Describes output of executing an `array` command on one node, including temporary files and socket numbers.	freed as part of a list
ascmdrsltlist_t	List of command results, one *ascmdrslt_t* per node where an `array` command was executed.	`asfreecmdrsltlist()`
asmachine_t	Configuration data about one node: machine name and attributes.	freed as part of a list
asmachinelist_t	List of *asmachine_t* structures, one per machine in the queried array	`asfreemachinelist()`
aspidlist_t	List of PID values.	`asfreepidlist()`

Error Message Conventions

The functions of the Array Services library have a complicated convention for error return codes. The man pages related to this convention are listed in Table 1-11.

Table 1-11 Error Message Functions

Function	Operation
`aserrorcode`(3X)	Discusses the error code conventions and some macro functions used to extract subfields from an error code.
`asmakeerror`(3X)	Constructs an error code value from its component parts.
`asstrerror`(3X)	Returns a descriptive string for a given error code value.
`asperror`(3X)	Prints a descriptive string, with a specified heading string, on stderr.

In general, each function sets a value in the global *aserrorcode*, which has type *aserror_t* (not necessarily an *int*). An error code is a structured value with these parts:

- *aserrno* is a general error number similar to those declared in `sys/errno.h`.
- *aserrwhy* documents the cause of the error.
- *aserrwhat* documents the component that detected the error.
- *aserrextra* may give additional information.

Macro functions to extract these subfields from the global *aserrorcode* are provided.

Connecting to Array Services Daemons

The functions listed in Table 1-12 are used to open a connection between the node where your program runs and an instance of `arrayd` in the same or another node.

Table 1-12 Functions for Connections to Array Services Daemons

Function	Operation
`asopenserver`(3X)	Establishes a logical connection to `arrayd` in a specified node, returning a token that represents that connection for use in other functions.
`ascloseserver`(3X)	Close an `arrayd` connection created by `asopenserver()`.
`asgetserveropt`(3X)	Return the local options currently in use by an instance of `arrayd`.
`asdfltserveropt`(3X)	Return the default options in effect at an instance of `arrayd`.
`assetserveropt`(3X)	Set new options for an instance of `arrayd`.

The key function is `asopenserver()`. It takes a nodename as a character string (as a user would give it in the `-s` option; see "Summary of Common Command Option", page 8), and optionally a socket number to override the default `arrayd` socket number. This function opens a socket connection to the specified instance of `arrayd`. The returned token (type *asserver_t*) stands for that connection and is passed to other functions.

The functions for getting and setting server options can change the configured options shown in Table 1-13. To set these options is the programmatic equivalent of passing command line options in an Array Services command (see "About Array Configuration", page 18 and "Using Array Services Commands", page 6).

Table 1-13 Server Options Functions Can Query or Change

Constant	Changeable?	Meaning
`AS_SO_TIMEOUT`	yes	Timeout interval for any request to this server.
`AS_SO_CTIMEOUT`	yes	Timeout interval for connecting to this server.
`AS_SO_FORWARD`	yes	Whether or not Array Services requests should be forwarded through the local `arrayd`, or sent directly (the `-F` option).

Constant	Changeable?	Meaning
AS_SO_LOCALKEY	yes	The local authentication key (the -Kl command option).
AS_SO_REMOTEKEY	yes	The remote authentication key (-Kr command option).
AS_SO_PORTNUM	no	In default options only, the default socket number.
AS_SO_HOSTNAME	no	The hostname for this connection.

Database Interrogation

The functions summarized in Table 1-14 are used to interrogate the configuration database used by arrayd in a specified node (see "About Array Configuration", page 18).

Table 1-14 Functions for Interrogating the Configuration

Function	Operation
asgetdfltarray(3X)	Return the array name and all attributes strings for the default array known to a specified server, in an *asarray_t* structure.
aslistarrays(3X)	Return the names of all arrays, with their attribute strings, from a specified server, as an *asarraylist_t* structure.

Function	Operation
`aslistmachines`(3X)	Return the names of all machines, with their attribute strings, from a specified server, as an *asmachinelist_t* structure.
`asgetattr`(3X)	Search for a particular attribute name in a list of attribute strings, and return its value.

Using these functions you can extract any arrayname, nodename, or attribute that is known to an `arrayd` instance you have opened.

Managing Array Service Handles

The functions summarized in Table 1-15 are used to create and interrogate ASH values.

Table 1-15 Functions for Managing Array Service Handles

Function	Operation
`asallocash`(3X)	Allocate a new ASH value. The value is only created, it is not applied to any process.
`aspidsinash`(3X)	Returns a list of PID values associated with an ASH at a specified server, as an *aspidlist_t* structure.
`asashofpid`(3X)	Returns the ASH associated with a specified PID.
`setash`(2)	Change the ASH of the calling process.

The `asallocash()` function is like the command `ainfo newash` (see "About Array Session Handles (ASH)", page 14). Only a program with root privilege can use the

`setash()` system function to change the ASH of the current process. Unprivileged processes can create new ASH values but cannot change their ASH.

The functions summarized in Table 1-16 are used to enumerate the active ASH values at a specified node. In each case, the list of ASH values is returned in an *asashlist_t* structure.

Table 1-16 Functions for ASH Interrogation

Function	Operation
`aslistashs`(3X)	Return active ASH values from one node or all nodes of a specified Array via a specified server.
`aslistashs_array`(3X)	Return active ASH values from an Array by name.
`aslistashs_server`(3X)	Return active ASH values known to a specified server node.
`aslistashs_local`(3X)	Return active ASH values in the local node.
`asashisglobal`(3X)	Test to see if an ASH is global.

Executing an array Command

The `ascommand()` function is the programmatic equivalent of the `array` command (see "Operation of Array Commands", page 24 and the `array(1)` man page). This command has many options and can be used to execute commands in three distinct modes.

The command to be executed must be prepared in an `ascmdreq_t` structure, which contains the following fields:

```
typedef struct ascmdreq {
   char *array;            /* Name of target array */
   int flags;              /* Option flags */
   int numargs;            /* Number of arguments */
   char **args;            /* Cmd arguments (ala argv) */
   int ioflags;            /* I/O flags for interactive commands */
   char rsrvd[100];        /* reserved for expansion: init to 0's */
} ascmdreq_t;
```

Your program must prepare this structure in order to execute a command. The option flags allow for the same controls as the command line options of `array`.

The result of the command is returned as an *ascmdrsltlist_t* structure, which is a vector of *ascmdrslt_t* structures, one for each node at which the command was executed. Each *ascmdrslt_t* contains the following fields:

```
typedef struct ascmdrslt {
    char     *machine;  /* Name of responding machine */
    ash_t    ash;       /* ASH of running command */
    int      flags;     /* Result flags */
    aserror_t error;    /* Error code for this command */
    int      status;    /* Exit status */
    char     *outfile;  /* Name of output file */
    int      ioflags;   /* I/O connections (see ascmdreq_t) */
    int      stdinfd;   /* File descriptor for command's stdin */
    int      stdoutfd;  /* File descriptor for command's stdout */
    int      stderrfd;  /* File descriptor for command's stderr */
    int      signalfd;  /* File descriptor for sending signals */
} ascmdrslt_t;
```

The fields *machine*, *ash*, *flags*, *error*, and *status* reflect the result of the command execution in that machine. The other fields depend on the mode of execution.

Normal Batch Execution

To execute a command in the normal way, waiting for it to complete and collecting its output, you do not set either `ASCMDREQ_NOWAIT` or `ASCMDREQ_INTERACTIVE` in the command option flags.

Control returns from `ascommand()` when the command is complete on all nodes. If the `ASCMDREQ_OUTPUT` flag was specified, and if the command definition does not specify a MERGE subentry (see "Summary of Command Definition Syntax", page 25),

the *outfile* result field contains the name of a temporary file containing one node's output stream.

When the command is implemented with a MERGE subentry, there is only one output file no matter how many nodes are invoked. In this case, the returned list contains only one *ascmdrslt_t* structure. It contains the ASCMDRSLT_MERGED and ASCMDREQ_OUTPUT flags, and the *outfile* result field contains the name of a temporary file containing the merged output.

Immediate Execution

When a command has no useful output and should execute concurrently with the calling program, you specify the ASCMDREQ_NOWAIT option. In this case, output cannot be collected because no program will be waiting to use it. Control returns as soon as the command has been distributed. The result structures do not reflect the command's result but only the result of trying to start it.

Interactive Execution

You can start a command in such a way that your program has direct interaction with the input and output streams of the command process in every node. When you do this, your program can supply input and inspect output in near real time.

To establish interactive execution, specify ASCMDREQ_INTERACTIVE in the command option flag. Also set one or more of the following flags in the *ioflags* field:

ASCMDIO_STDIN	Requests a socket attached to the command's stdin.
ASCMDIO_STDOUT	Requests a socket attached to the command's stdout.
ASCMDIO_STDERR	Requests a socket attached to the command's stderr.
ASCMDIO_SIGNAL	Requests a socket that can be used to deliver signals.

As with ASCMDREQ_NOWAIT, control returns as soon as the command has been distributed. Each result structure contains file descriptors for the requested sockets for the command process in that node.

Your program writes data into the *stdinfd* file descriptor of one node in order to send data to the stdin stream in that node. Your program reads data from the *stdoutfd* file descriptor to read one node's output stream.

You will typically use either the `select()` or the `poll()` system function to learn when one of the sockets is ready for use. You may choose to start one or more subprocesses using `fork()` to handle I/O to the sockets of each node (see the `select`(2) , `poll`(2) and `sproc`(2) man pages). (You may also use `sproc()` to make subprocesses, but keep in mind that the libarray is not thread-safe, so it should only be used from one process in a share group.)

Executing a User Command

The `asrcmd()` function allows a program to initiate any user command string on a specified node. This provides a powerful facility for remote execution that does not require root privilege, as the standard rcmd() function does (compare the `asrcmd`(3) and `rcmd`(3) man pages).

The `asrcmd()` function takes arguments specifying:

- The array node to use, as returned by `asopenserver()` (see "Connecting to Array Services Daemons", page 32).
- The user name to use on the remote node.
- The command line to be executed.

The returned value (as with `rcmd()`) is a socket that represents the standard input and output streams of the executing command. Optionally, a separate socket for the standard error stream can be obtained.

Comprehensive System Accounting

Comprehensive System Accounting (CSA) provides detailed, accurate accounting data per job. It also provides data from some daemons. CSA is dependent on the concept of a Linux kernel job which is described in detail in the following paragraphs.

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 Linux 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 41, shows the point of entry processes that initiate the creation of jobs.

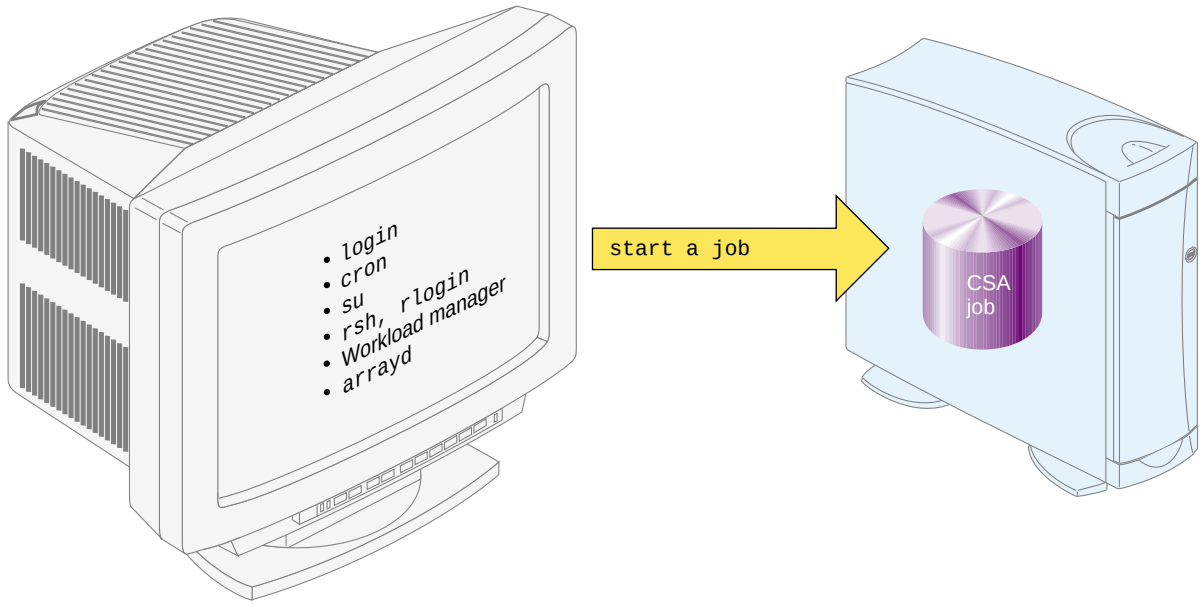

Figure 2-1 Point of Entry Processes

A Linux job has the following characteristics:

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

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

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

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

This chapter contains the following sections:

CSA Overview

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

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

CSA takes this per-process accounting information and combines it by job identifier (`jid`) within system boot uptime periods. CSA accounting for a job consists of all accounting data for a given job identifier during a single system boot period.

However, since workload management jobs may span multiple reboots and thereby consist of multiple job identifiers, CSA accounting for these jobs includes the accounting data associated with the workload management identifier.

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

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

A system billing unit (SBU) is a unit of measure that reflects use of machine resources. SBUs are defined in the CSA configuration file `/etc/csa.conf` and are set to `0.0` by default. The weighting factor associated with each field in the CSA accounting records can be altered to obtain an SBU value suitable for your site. For more information on SBUs, see "System Billing Units (SBUs)", page 76.

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

There are four user exits available with the `csarun`(8) daily accounting script. There is one user exit available with the `csaperiod`(8) monthly accounting script. These user exits allow sites to tailor the daily and monthly run of accounting to their specific needs by creating user exit scripts to perform any additional processing and to allow archiving of accounting data. See the `csarun`(8) and `csaperiod`(8) man pages for further information.

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

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

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 47.

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 CSA, daily accounting can be run as many times as necessary during a day; however, this feature is still referred to as daily accounting.
Job	A job is a grouping of processes that the system treats as a single entity and is identified by a unique job identifier (job ID). CSA is the only accounting type to organize accounting data by jobs and boot times and then place the data into a `sorted pacct` file. For non-workload management jobs, a job consists of all accounting data for a given job ID during a single boot period. A workload management job consists of the accounting data for all job IDs associated with the job's workload management request ID. Workload management jobs may span multiple boot periods. If a job is restarted, it has the same job ID associated with it during all boot periods in which it runs. Rerun workload management jobs have multiple job IDs. CSA treats all phases of a workload management job as being in the same job.
Periodic accounting	Periodic (monthly) accounting further processes, reports, and summarizes the daily accounting reports to give a higher level view of how the system is being used. CSA lets system administrators specify the time periods for which monthly or cumulative accounting is to be run. Thus, periodic accounting can be run more than

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

The following abbreviations and definitions are used throughout this chapter:

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

Enabling or Disabling CSA

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

1. Install Linux CSA using the instructions in the `README.csa` file from the download link at:

   ```
   http://oss.sgi.com/projects/csa
   ```

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

   ```
   chkconfig --add csaacct
   ```

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

4. Use the `csaswitch`(8) 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`(8) 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 57.

The following steps are required to disable CSA job accounting:

1. To turn off CSA, use the `csaswitch`(8) command:

   ```
   csaswitch -c halt
   ```

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

   ```
   chkconfig --del csaacct
   ```

CSA Files and Directories

The following sections describe the CSA files and directories.

Files in the `/var/csa` Directory

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

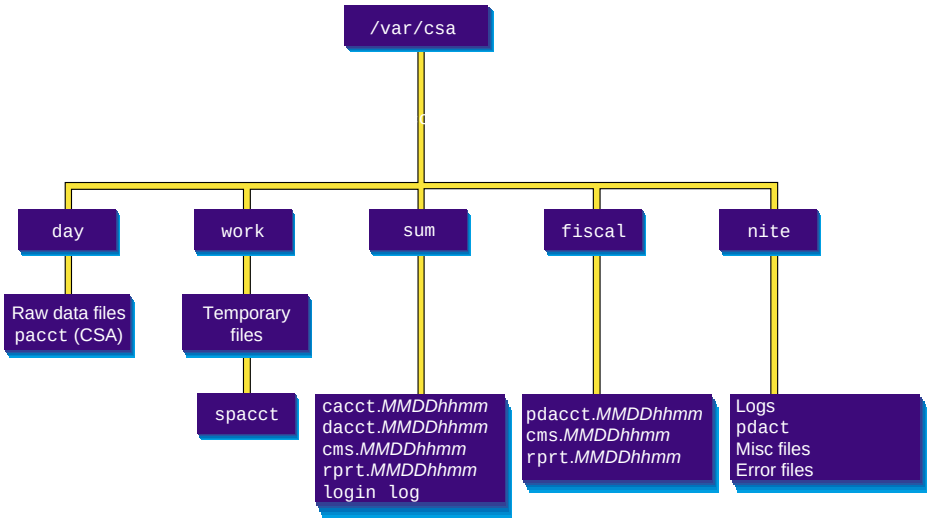

Figure 2-2 The `/var/csa` Directory

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

Files in the `/var/csa/` Directory

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

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

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

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

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

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

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

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

File	Description
`BAD.Wpacct*`	Unprocessed accounting data containing invalid records (verified by `csaverify`(8)).
`Ever.tmp1`	Data verification work file.
`Ever.tmp2`	Data verification work file.
`Rpacct0`	Process and daemon accounting data to be recycled in the next accounting run.
`Wdiskcacct`	Disk accounting data (`cacct.h` format) created by `dodisk`(8) (See the `dodisk`(8) man page).
`Wdtmp`	Disk accounting data (ASCII) created by `dodisk`(8).
`Wpacct*`	Raw process and daemon accounting data.

`spacct`	`sorted pacct` file

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

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

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

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

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

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

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

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

File	Description
`active`	Used by the `csarun`(8) command to record progress and print warning and error messages. `active`*MMDDhhmm* is the same as `active` after `csarun` detects an error.

`clastdate`	Last two times `csarun` was executed; in *MMDDhhmm* format.
`dk2log`	Diagnostic output created during execution of `dodisk` (see the `cron` entry for `dodisk` in "Setting Up CSA", page 57).
`diskcacct`	Disk accounting records in `cacct.h` format, created by `dodisk`.
`Eaddc`*MMDDhhmm*	Error/warning messages from the `csaaddc`(8) command for an accounting run done on *MMDD* at *hhmm*.
`Earc1`*MMDDhhmm*	Error/warning messages from the `csa.archive1`(8) command for an accounting run done on *MMDD* at *hhmm*.
`Earc2`*MMDDhhmm*	Error/warning messages from the `csa.archive2`(8) command for an accounting run done on *MMDD* at *hhmm*.
`Ebld.`*MMDDhhmm*	Error/warning messages from the `csabuild`(8) command for an accounting run done on *MMDD* at *hhmm*.
`Ecmd.`*MMDDhhmm*	Error/warning messages from the `csacms`(8) command when generating an ASCII report for an accounting run done on *MMDD* at *hhmm*.
`Ecms.`*MMDDhhmm*	Error/warning messages from the `csacms`(8) command when generating binary data for an accounting run done on *MMDD* at *hhmm*.
`Econ.`*MMDDhhmm*	Error/warning messages from the `csacon`(8) command for an accounting run done on *MMDD* at *hhmm*.
`Ecrep.`*MMDDhhmm*	Error/warning messages from the `csacrep`(8) command for an accounting run done on *MMDD* at *hhmm*.
`Ecrpt.`*MMDDhhmm*	Error/warning messages from the `csacrep`(8) command for an accounting run done on *MMDD* at *hhmm*.
`Edrpt.`*MMDDhhmm*	Error/warning messages from the `csadrep`(8) command for an accounting run done on *MMDD* at *hhmm*.

`Erec.`*MMDDhhmm*	Error/warning messages from the `csarecy`(8) command for an accounting run done on *MMDD* at *hhmm*.
`Euser.`*MMDDhhmm*	Error/warning messages from the `csa.user`(8) user exit for an accounting run done on *MMDD* at *hhmm*.
`Epuser.`*MMDDhhmm*	Error/warning messages from the `csa.puser`(8) user exit for an accounting run done on *MMDD* at *hhmm*.
`Ever.tmp1`*MMDDhhmm*	Output file from invalid record offsets from the `csaverify`(8) command for an accounting run done on *MMDD* at *hhmm*.
`Ever.tmp2`*MMDDhhmm*	Error/warning messages from the `csaverify`(8) command for an accounting run done on *MMDD* at *hhmm*.
`Ever.`*MMDDhhmm*	Error/warning messages from the `csaedit`(8) and `csaverify`(8) command (from the `Ever.tmp2` file) for an accounting run done on *MMDD* at *hhmm*.
`fd2log`	Diagnostic output created during execution of `csarun` (see `cron` entry for `csarun` in "Setting Up CSA", page 57).
`lock lock1`	Used to control serial use of the `csarun`(8) comand.
`pd2log`	Diagnostic output created during execution of `csaperiod` (see `cron` entry for `csaperiod` in "Setting Up CSA", page 57).
`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/local/sbin Directory

The /usr/local/sbin 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/local/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/local/sbin 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/rc.d Directory

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

Comprehensive System Accounting Expanded Description

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

Daily Operation Overview

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

1. When CSA accounting is enabled and the system is switched to multiuser mode, the `/usr/local/sbin/csaswitch` (see the `csaswitch`(8) man page) command is called by `local/sbin/rc.d/init.d/CSAAcct`.
2. By default, `csa`, memory, and I/O record types are enabled in `/etc/csa.conf`. However, to run workload management and tape daemon accounting, you must modify the `/etc/csa.conf` and the appropriate subsystem. For more information, see "Setting Up CSA", page 57.
3. The amount of disk space used by each user is determined periodically. The `/usr/local/sbin/dodisk` command (see `dodisk`(8)) is run periodically by the `cron` command to generate a snapshot of the amount of disk space being used by each user. The `dodisk` command should be run at most once for each time `/usr/local/sbin/csarun` is run (see `csarun`(8)). Multiple invocations of `dodisk` during the same accounting period write over previous `dodisk` output.
4. A fee file is created. Sites desiring to charge fees to certain users can do so by invoking `/usr/local/sbin/csachargefee` (see `csachargefee`(8)). Each accounting period's fee file (`/var/csa/day/fee`) is merged into the consolidated accounting records by `/usr/local/sbin/csaperiod` (see `csaperiod`(8)).
5. Daily accounting is run. At specified times during the day, `csarun` is executed by the `cron` command to process the current accounting data. The output from `csarun` is daily accounting files and an ASCII report.
6. Periodic (monthly) accounting is run. At a specific time during the day, or on certain days of the month, `/usr/local/sbin/csaperiod` (see `csaperiod`) is executed by the `cron` command to process consolidated accounting data from previous accounting periods. The output from `csaperiod` is periodic (monthly) accounting files and an ASCII report.

7. Accounting is disabled. When the system is shut down gracefully, the `csaswitch`(8) command is executed to halt all CSA process and daemon accounting.

Setting Up CSA

The following is a brief description of setting up CSA. Site-specific modifications are discussed in detail in "Tailoring CSA", page 76. As described in this section, CSA is run by a person with superuser permissions.

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

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

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

These entries are described in the following steps:

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

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

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

The `dodisk` command must be invoked with the `-c` option. For more information, see the `dodisk`(8) man page.

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

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

The `cron` command should periodically execute the `csackpacct`(8) shell script. If the `pacct` file grows larger than 4000 1K blocks (default), `csackpacct` calls the command `/usr/local/sbin/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/csa`. If there are not enough blocks, CSA accounting is turned off. The next time `csackpacct` is executed, it turns CSA accounting back on if there are enough free blocks.

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

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

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

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

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

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

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

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

automatically substituted for the year. The following are examples of two valid entries:

```
YEAR        1995
YEAR        *
```

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

  period prime_time_start nonprime_time_start

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

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

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

    ```
    WEEKDAY   0730    0001
    ```

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

    ```
    WEEKDAY   0730    1730
    ```

  – NONE/ALL or ALL/NONE. These start times specify that the entire period is to be either all prime time or all nonprime time. To specify that the entire period is to be considered prime time, set *prime_time_start* to ALL and *nonprime_time_start* to NONE. If the period is to be considered all nonprime time, set *prime_time_start* to NONE and *nonprime_time_start* to ALL. For example, to specify Monday through Friday as all prime time, you would enter the following:

    ```
    WEEKDAY ALL NONE
    ```

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

    ```
    SUNDAY NONE ALL
    ```

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

  day-of-year Month Day Description of Holiday

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

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

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

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

  The defaults values are as follows:

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

The `csarun` Command

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

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

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

Daily Invocation

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

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

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

Before executing `csarun` in this manner, ensure that the previous invocation completed successfully. To do this, look at the files `active` and `statefile` in `/var/csa/nite`. Both files should specify that the last invocation completed successfully. See "Restarting `csarun`", page 64.

Error and Status Messages

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

Errors and warning messages from programs called by `csarun` are written to files that have names beginning with `E` and ending with the current date and time. For example, `Ebld.11121400` is an error file from `csabuild` for a `csarun` invocation on November 12, at 14:00.

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

States

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

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

State	Description
`SETUP`	The current accounting file is switched via `csaswitch`. The accounting file is then moved to the `/var/csa/work/`*MMDD/hhmm* directory. File names are prefaced with `W`. `/var/csa/nite/diskcacct` is also moved to this directory.
`VERIFY`	The accounting files are checked for valid data. Records with invalid data are removed. Names of bad data files are prefixed with `BAD.` in the `/var/csa/work/`*MMDD/hhmm* directory. The corrected files do not have this prefix.
`ARCHIVE1`	First user exit of the `csarun` script. If a script named `/usr/local/sbin/csa.archive1` exists, it will be executed through the shell . (dot) command. The . (dot) command will not execute a compiled program, but the user exit script can. You might use this user exit to archive the accounting files in `${WORK}`.
`BUILD`	The `pacct` accounting data is organized into a `sorted pacct` file.
`ARCHIVE2`	Second user exit of the `csarun` script. If a script named `/usr/local/sbin/csa.archive2` exists, it will be executed through the shell . (dot) command. The . (dot) command will not execute a compiled program, but the user exit script can. You might use this exit to archive the `sorted pacct` file.
`CMS`	Produces a command summary file in `cms.h` format. The `cms` file is written to `/var/csa/sum/cms.`*MMDDhhmm* for use by `csaperiod`.
`REPORT`	Generates the daily accounting report and puts it into `/var/csa/sum/rprt.`*MMDDhhmm*. A consolidated data file, `/var/csa/sum/cacct.`*MMDDhhmm*, is also produced from the `sorted pacct` file. In addition, accounting data for unfinished jobs is recycled.
`DREP`	Generates a daemon usage report based on the `sorted pacct` file. This report is appended to the daily accounting report, `/var/csa/sum/rprt.`*MMDDhhmm*.
`FEF`	Third user exit of the `csarun` script. If a script named `/var/local/sbin/csa.fef` exists, it will be executed through the

shell . (dot) command. The . (dot) command will not execute a compiled program, but the user exit script can. The `csarun` variables are available, without being exported, to the user exit script. You might use this exit to convert the `sorted pacct` file to a format suitable for a front-end system.

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

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

Restarting csarun

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

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

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

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

To restart `csarun`, follow these steps:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Verifying and Editing Data Files

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

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

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

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

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

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

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

   `/usr/local/sbin/csaverify -P pacct.NEW`

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

CSA Data Processing

The flow of data among the various CSA programs is explained in this section and is illustrated in Figure 2-3.

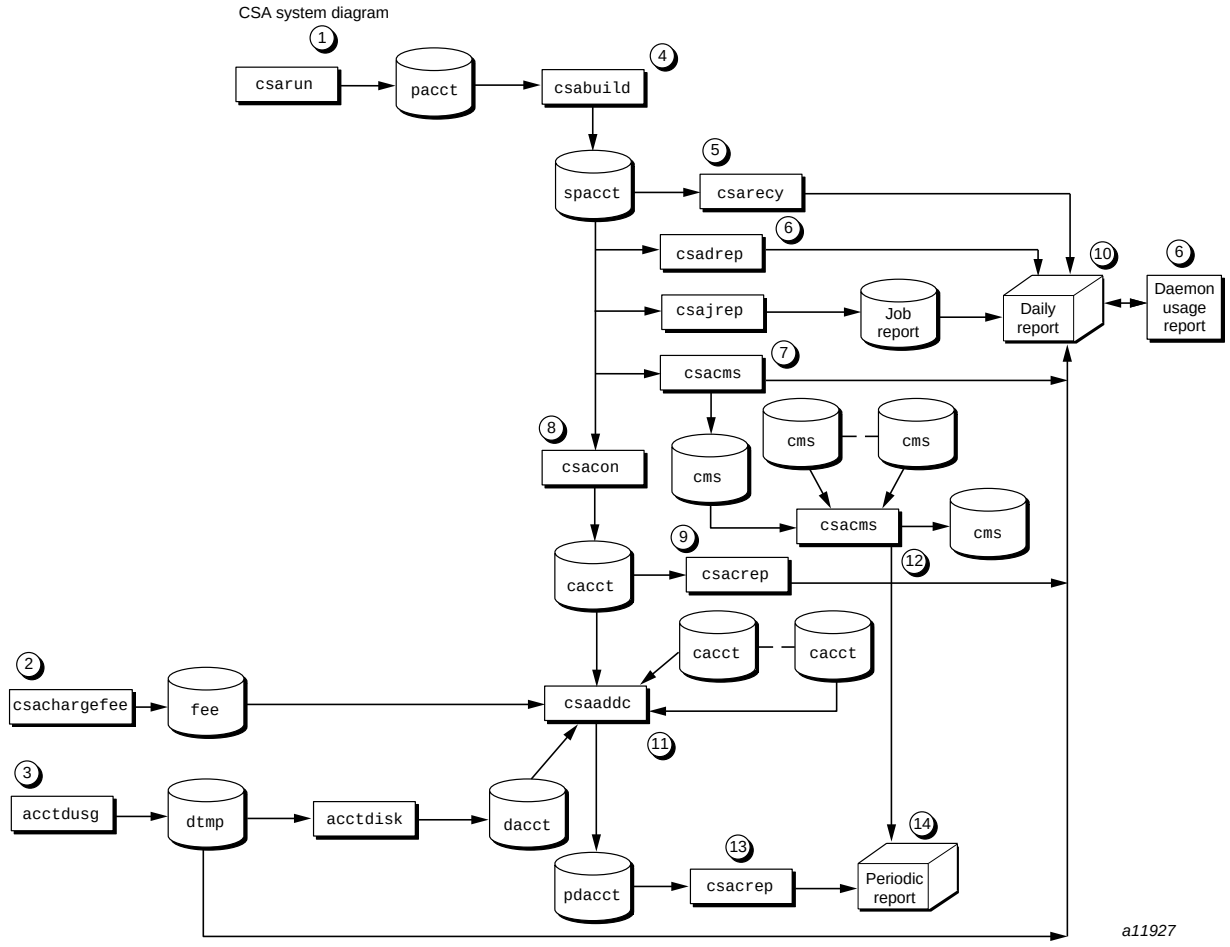

Figure 2-3 CSA Data Processing

1. Generate raw accounting files. Various daemons and system processes write to the raw `pacct` accounting files.
2. Create a fee file. Sites that want to charge fees to certain users can do so with the `csachargefee`(8) command. The `csachargefee` command creates a fee file that is processed by `csaaddc`(8).

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

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

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

   Sometimes data for terminated jobs are continually recycled. This can occur when accounting data is lost. To prevent data from recycling forever, edit `csarun` so that `csabuild` is executed with the `-o` *nday* option, which causes all jobs older than *nday* days to terminate. Select an appropriate *nday* value (see the `csabuild` man page for more information and "Data Recycling", page 70).

6. Generate the daemon usage report, which is appended to the daily report. `csadrep`(8) reports usage of the workload management and tape (tape is deferred) daemons. Input is either from a `sorted pacct` file created by `csabuild`(8) or from a binary file created by `csadrep` with the `-o` option. The files operand specifies the binary files.

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

8. Condense records from the `sorted pacct` file. The `csacon`(8) command condenses records from the `sorted pacct` file and writes consolidated records in `cacct` format to `var/csa/sum/cacct.`*MMDDhhmm*.

9. Generate an accounting report based on the consolidated data. The `csacrep`(8) command generates reports from data in `cacct` format, such as output from the `csacon`(8) command. The report format is determined by the value of `CSACREP` in the `/etc/csa.conf` file. Unless modified, it will report the CPU time, total `KCORE` minutes total `KVIRTUAL` minutes, block I/O wait time, and raw I/O wait time. The report will be sorted first by user ID and then by the secondary key of project ID (project ID is deferred) and the headers will be printed.
10. Create the daily accounting report. The daily accounting report includes the following:
    - Consolidated information report (step 11)
    - Unfinished recycled jobs (step 5)
    - Disk usage report (step 3)
    - Daily command summary (step 7)
    - Last login information
    - Daemon usage report (step 6)
11. Combine `cacct` records. The `csaaddc`(8) command combines `cacct` records by specified consolidation options and writes out a consolidated record in `cacct` format.
12. Summarize command usage from per-process accounting records. The `csacms`(8) command reads the cms files created in step 7. Both an ASCII and a binary file are created.
13. Produce a consolidated accounting report. `csacrep`(8) is used to generate a report based on a periodic accounting file.
14. The periodic accounting report layout is as follows:
    - Consolidated information report
    - Command summary report

Steps 4 through 11 are performed during each accounting period by `csarun`(8). Periodic (monthly) accounting (steps 12 through 14) is initiated by the `csaperiod`(8) command. Daily and periodic accounting, as well as fee and disk usage generation (steps 2 through 3), can be scheduled by `cron`(8) to execute regularly. See "Setting Up CSA", page 57, 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/csa/day/pacct0` file.

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

How Jobs Are Terminated

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

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

Unlike interactive jobs, workload management requests can have multiple `EOJ` records associated with them. In addition to the request's `EOJ` record, there can be `EOJ` records for net clients and checkpointed portions of the request. The net client perform workload management processing on behalf of the request.

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

- The job is an interactive, `cron`, or `at` job, and there is an `EOJ` record for the job in the `pacct` file.

- The job is a workload management request, and there is both an `EOJ` record for the request and an `NQ_DISP` record type in the `pacct` file.
- The job is an interactive, `cron`, or `at` job and is active at the time of a system crash.
- The job is manually terminated by the administrator using one of the methods described in "How to Remove Recycled Data", page 71.

Why Recycled Sessions Should Be Scrutinized

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

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

- Kernel or daemon accounting is turned off.

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

How to Remove Recycled Data

Before choosing to delete recycled data, you should understand the repercussions, as described in "Adverse Effects of Removing Recycled Data", page 73. 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/local/sbin/csarun`. Change the `-r` option on the `csarecy` invocation line to `-A`. Also, do not redirect standard output to `${SUM_DIR}/recyrpt`. The result should be similar to the following:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Adverse Effects of Removing Recycled Data

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

- Users may or may not be billed for terminated recycled jobs. Administrators must understand which of the previously described methods cause the user to be billed for the terminated recycled jobs. It is up to the site to decide whether or not it is valid for the user to be billed for these jobs.

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

- It may be impossible to reconstruct a terminated recycled job. If a recycled job is terminated by the administrator, but the job actually terminates in a later accounting period, information about the job is lost. If a user questions the

resource billing, it may be extremely difficult or impossible for the administrator to correctly reassemble all accounting information for the job in question.

- Manually terminated recycled jobs may be improperly billed in a future billing period. If the accounting data for the first portion of a job has been deleted, CSA may be unable to correctly identify the remaining portion of the job. Errors may occur, such as workload management requests being flagged as interactive jobs, or workload management requests being billed at the wrong queue rate. This is explained in detail in "Workload Management Requests and Recycled Data", page 75.
- 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 71.

Table 2-1 Possible Effects of Removing Recycled Data

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

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

Workload Management Requests and Recycled Data

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

- CSA fails to flag the job as a workload management job. This causes the request to be billed at standard rates instead of aworkload management queue rate (see "Workload Management SBUs", page 79.
- The request is billed at the wrong queue rate.
- The wrong queue wait time is associated with the request.

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

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

- The workload management daemon receives a request.
- A request executes. This includes executing a request for the first time, restarting, and rerunning a request.
- A request terminates. A workload management request can terminate because it is completed, requeued, held, rerun, or migrated.
- Output is delivered.

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

Tailoring CSA

This section describes the following actions in CSA:

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

System Billing Units (SBUs)

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

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

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

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

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

then

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

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

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

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

Process SBUs

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

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

Value	Description
`P_BASIC`	Prime-time weight factor. `P_BASIC` is multiplied by the sum of prime time SBU values to get the final SBU factor for the process record.
`P_TIME`	General-time weight factor. `P_TIME` is multiplied by the time SBUs (made up of `P_STIME`, `P_UTIME`, `P_QTIME`, `P_BWTIME`, and `P_RWTIME`) to get the time contribution to the process record SBU value.
`P_STIME`	System CPU-time weight factor. The unit used for this weight is *billing units* per second. `P_STIME` is multiplied by the system CPU time.
`P_UTIME`	User CPU-time weight factor. The unit used for this weight is *billing units* per second. `P_UTIME` is multiplied by the user CPU time.

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

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

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

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

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

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

The variables in this formula are described as follows:

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

Workload Management SBUs

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

The `WKMG_NUM_QUEUES` parameter sets the number of queues for which you want to set SBUs (the value must be set to at least 1). Each `WKMG_QUEUE` *x* variable in the configuration file has a queue name and an SBU pair associated with it (the total

number of queue/SBU pairs must equal WKMG_NUM_QUEUES). The queue/SBU pairs define weights for the queues. If an SBU value is less than 1.0, there is an incentive to run jobs in the associated queue; if the value is 1.0, jobs are charged as though they are non-workload management jobs; and if the SBU is 0.0, there is no charge for jobs running in the associated queue. SBUs for queues not found in the configuration file are automatically set to 1.0.

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

Tape SBUs (deferred)

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

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

Note: Tape support is deferred.

Daemon Accounting

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

In most cases, daemon accounting must be enabled by both the CSA subsystem and the daemon. "Setting Up CSA", page 57, 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/local/sbin/csaswitch -c on -n tape
```

Daemon accounting is disabled at system shutdown (see "Setting Up CSA", page 57). It can also be disabled at any time by the `csaswitch` command when used with the `off` operand. For example, to disable workload management accounting, execute the following command:

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

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

Setting up User Exits

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

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

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

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

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

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

If the script exists, it is executed via the shell . (dot) command. If the script does not exist, the user exit is ignored. The . (dot) command will not execute a compiled program, but the user exit script can. `csarun` variables are available, without being

exported, to the user exit script. `csarun` checks the return status from the user exit and if it is nonzero, the execution of `csarun` is terminated.

Charging for Workload Management Jobs

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

The following table describes the termination codes:

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

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

Tailoring CSA Shell Scripts and Commands

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

Variable	Description
`ACCT_FS`	File system on which `/var/csa` resides. The default is `/var`.
`MAIL_LIST`	List of users to whom mail is sent if fatal errors are detected in the accounting shell scripts. The default is `root` and `adm`.

WMAIL_LIST	List of users to whom mail is sent if warning errors are detected by the accounting scripts at cleanup time. The default is root and adm.
MIN_BLKS	Minimum number of free blocks needed in ${ACCT_FS} to run csarun or csaperiod. The default is 2000 free blocks. Block size is 1024 bytes.

Using at to Execute csarun

You can use the at command instead of cron to execute csarun periodically. If your system is down when csarun is scheduled to run via cron, csarun will not be executed until the next scheduled time. On the other hand, at jobs execute when the machine reboots if their scheduled execution time was during a down period.

You can execute csarun by using at in several ways. For example, a separate script can be written to execute csarun and then resubmit the job at a specified time. Also, an at invocation of csarun could be placed in a user exit script, /usr/local/sbin/csa.user, that is executed from the USEREXIT section of csarun. For more information, see "Setting up User Exits", page 81.

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/local/sbin/csarun 2> /var/csa/nite/fd2log
```

CSA Reports

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

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

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

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

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

This section describes the following reports:

CSA Daily Report

This section describes the following reports:

Consolidated Information Report

The Consolidated Information Report is sorted by user ID and then project ID (project ID is deferred). 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 (deferred)
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 (deferred)
STARTED	Beginning time of this job

Disk Usage Report

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

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

Command Summary Report

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

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

Last Login Report

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

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

Daemon Usage Report

Daemon Usage Report shows reports usage of the workload management and tape daemons (tape is deferred). This report has several individual reports depending upon if there was workload management or tape daemon activity within this reporting period.

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

Heading	Description
Job Type	Type of job (interactive or workload management)
Total Job Count	Number and percentage of jobs per job type
Tape Jobs	Number and percentage of tape jobs associated with these interactive and workload management job (deferred)

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

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

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

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

Periodic Report

This section describes two periodic reports as follows:

- "Consolidated accounting report", page 88
- "Command summary report", page 89

Consolidated accounting report

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

Heading	Description
PROJECT NAME	Project associated with this resource usage information
USER ID	User identifier
LOGIN NAME	Login name for the user identifier
CPU_TIME	Total accumulated CPU time in seconds
KCORE * CPU-MIN	Total accumulated amount of Kbytes of core (physical) memory used per minute of CPU time of processes
KVIRT * CPU-MIN	Total accumulated amount of Kbytes of virtual memory used per minute of CPU time
IOWAIT BLOCK	Total accumulated block I/O wait time in seconds
IOWAIT RAW	Total accumulated raw I/O wait time in seconds
DISK BLOCKS	Total number of disk blocks used

DISK SAMPLES	Number of times disk accounting was run to obtain the disk blocks used value
FEE	Total fees charged to this user from `csachargefee`(8)
SBUs	System billing units charged to this user and project

Command summary report

The following information summarizes command usage during the defined reporting period. The usage values are the total amount of resources used by all invocations of the specified command. Unlike the daily command summary report, the periodic command summary report displays all command entries. Commands executed only once are not combined together into an "***other" entry but are listed individually in the periodic command summary report.

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

BLOCKS READ	Total number of blocks read
BLOCKS WRITTEN	Total number of blocks written

CSA Man Pages

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

User-Level Man Pages

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

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

Administrator Man Pages

The following administrator man page is provided with CSA software:

Administrator man page	Description
`csaaddc`(8)	Combines `cacct` records.
`csabuild`(8)	Organizes accounting records into job records.
`csachargefee`(8)	Charges a fee to a user.
`csackpacct`(8)	Checks the size of the CSA process accounting file.
`csacms`(8)	Summarizes command usage from per-process accounting records
`csacon`(8)	Condenses records from the `sorted pacct` file.

`csacrep`(8)	Reports on consolidated accounting data.
`csadrep`(8)	Reports daemon usage.
`csaedit`(8)	Displays and edits the accounting information.
`csagetconfig`(8)	Searches the accounting configuration file for the specified argument.
`csajrep`(8)	Prints a job report from the `sorted pacct` file.
`csarecy`(8)	Recycles unfinished jobs into the next accounting run.
`csaswitch`(8)	Checks the status of, enables or disables the different types of CSA, and switches accounting files for maintainability.
`csaverify`(8)	Verifies that the accounting records are valid.

Index