



# 4

## Informational Tools

---

### 4.1 Chapter Objectives

The objective of this chapter is to introduce the tools that are available to assist in diagnosing and determining system configuration, development, and performance problems. We will use some of these tools later in the book to examine particular situations.

By the end of the chapter, the reader will have knowledge of most commonly useful tools available both on Solaris and as part of the developer tools.

### 4.2 Tools That Report System Configuration

#### 4.2.1 Introduction

This section covers tools that report static information about the system, such as the type of processor installed and so forth.

#### 4.2.2 Reporting General System Information (`prtdiag`, `prtconf`, `prtpicl`, `prtfwu`)

`prtdiag` is a purely informational tool that prints the machine's diagnostic details. The exact output depends on the system. I included it in this chapter

because it can be a source of useful information about the setup of the machine. It is located in `/usr/sbin`. The tool is often the first place to look for information on the processors and memory that are installed in the system, as well as general system configuration.

Output from `prtdiag` on a two-CPU UltraSPARC IIIc system is shown in Example 4.1. The output identifies the processors and the system clock speed, together with the amount of memory installed and in which memory slots it is installed. The output refers to the UltraSPARC IIIc processor as the UltraSPARC III+, and further abbreviates this to US-3+.

### Example 4.1 Sample Output from `prtdiag`

```

$ /usr/sbin/prtdiag
System Configuration: Sun Microsystems sun4u SUNW,Sun-Blade-1000 (2xUltraSPARC III+)
System clock frequency: 150 MHz
Memory size: 2GB
===== CPUs =====
CPU   Freq      E$      CPU      CPU      Temperature
      Size      Impl.   Mask     Die      Ambient
-----
  0    900 MHz  8MB     US-III+  2.2     75 C    25 C
  1    900 MHz  8MB     US-III+  2.2     75 C    24 C
===== IO Devices =====
Brd   Bus   Freq  Slot  Name                                     Model
-----
  0   pci   33    1    SUNW,m64B (display)                     SUNW,370-4362
  0   pci   66    4    SUNW,qlc-pci1077,2200.5 (scsi-fc+)
  0   pci   33    5    ebus/parallel-ns87317-ecpp (para+
  0   pci   33    5    ebus/serial-sab82532 (serial)
  0   pci   33    5    network-pci108e,1101.1 (network)       SUNW,pci-eri
  0   pci   33    5    firewire-pci108e,1102.1001 (fire+
  0   pci   33    6    scsi-pci1000,f.37 (scsi-2)
  0   pci   33    6    scsi-pci1000,f.37 (scsi-2)
===== Memory Configuration =====
Segment Table:
-----
Base Address      Size      Interleave Factor  Contains
-----
0x0                2GB           4                BankIDs 0,1,2,3
Bank Table:
-----
Physical Location
ID      ControllerID  GroupID  Size      Interleave Way
-----
  0         0           0      512MB     0
  1         0           1      512MB     1
  2         0           0      512MB     2
  3         0           1      512MB     3
Memory Module Groups:
-----
ControllerID  GroupID  Labels
-----
  0             0      J0100,J0202,J0304,J0406
  0             1      J0101,J0203,J0305,J0407

```

Other tools exist that provide system information at various levels of detail. The tools `prtconf`, `prtpicl`, and `prtfru` produce long lists of system configuration information, the contents of which depend on the details available on the particular platform.

### 4.2.3 Enabling Virtual Processors (`psrinfo` and `psradm`)

`psrinfo` is a tool that will report whether the virtual processors are enabled. Sample output from `psrinfo`, run on a system with two 900MHz processors, is shown in Example 4.2. You can obtain more detailed output using `psrinfo -v`.

**Example 4.2** Sample Output from `psrinfo` and `psrinfo -v`

```
$ psrinfo
0      on-line   since 11/20/2003 11:18:59
1      on-line   since 11/20/2003 11:19:00
$ psrinfo -v
Status of virtual processor 0 as of: 10/23/2006 21:47:30
on-line since 11/20/2003 11:19:00.
The sparcv9 processor operates at 900 MHz,
and has a sparcv9 floating-point processor.
Status of virtual processor 1 as of: 10/23/2006 21:47:30
on-line since 11/20/2003 11:19:00.
The sparcv9 processor operates at 900 MHz,
and has a sparcv9 floating-point processor.
```

Solaris 10 introduced the `-p` option to `psrinfo` that reports on the physical processors in the system. Example 4.3 shows the output from a system that has a single UltraSPARC T1 physical processor with 32 virtual processors.

**Example 4.3** Output from `psrinfo -pv` from an UltraSPARC T1 System

```
$ psrinfo -pv
The physical processor has 32 virtual processors (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11,
12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31)
UltraSPARC T1 (cpuid 0 clock 1200 MHz)
```

You can enable or disable the virtual processors using the `psradm` tool. The `-f` flag will disable a processor and the `-n` flag will enable it. This tool is available only with superuser permissions.

The `-i` flag for `psradm` excludes CPUs from handling interrupts; this may be of use when partitioning the workload over multiple CPUs. Example 4.4 shows the command for excluding CPU number 1 from the CPUs that are available to handle interrupts.

### Example 4.4 Excluding a CPU from Interrupt Handling

```
$ psradm -i 1
$
```

## 4.2.4 Controlling the Use of Processors through Processor Sets or Binding (`psrset` and `pbind`)

It is possible to configure systems so that the processors are kept in discrete sets. This will partition compute resources so that particular applications run on particular sets of processors. The command to do this is `psrset`, and it is available only with superuser permissions. Example 4.5 illustrates the use of processor sets.

### Example 4.5 Example of the `psrset` Command

```
# psrset -c 1
created processor set 1
processor 1: was not assigned, now 1
# psrset
user processor set 1: processor 1
# psrset -e 1 sleep 1
# psrset -d 1
removed processor set 1
```

The code in Example 4.5 first shows the creation of a processor set using the `psrset -c` option, which takes a list of processor IDs and binds those processors into a set. The command returns the `id` of the set that has just been created. The command `psrset` with no options reports the processor sets that are currently in existence, and the processors that belong to those sets. It is possible to run a particular process on a given set using the `psrset -e` option, which takes both the processor set to use and the command to execute on that set. Finally, the `psrset -d` option deletes the processor set that is specified.

You must be careful when using processor sets (or any partitioning of the processor resources). Using processor sets, it is possible to introduce *load imbalance*, in which a set of processors is oversubscribed while another set is idle. You need to consider the allocation of processors to sets at the level of the entire machine, which is why the command requires superuser privileges.

It is a good practice to check for both the number of enabled virtual processors (using `psrinfo`) and the existence of processor sets whenever the system's performance is being investigated. On systems where processor sets are used regularly, or processors are often switched off, they can be a common reason for the system not providing the expected performance.

It is also possible to bind a single process to a particular processor using the `pbind` command, which takes the `-b` flag together with the `pid` and the processor ID as inputs when binding a process to a processor, and the `-u` flag together with the `pid` to unbind the process. Unlike processor sets that exclude other processes from running on a given group of processors, processor binding ensures that a particular process will run on a particular processor, but it does not ensure that other processes will not also run there.

### 4.2.5 Reporting Instruction Sets Supported by Hardware (`isalist`)

`isalist` is a Solaris tool that outputs the instruction sets architectures (ISAs) the processor supports. This can be useful for picking the appropriate compiler options (this will be covered in Section 5.6.5 of Chapter 5). It is also useful in determining the particular variant of CPU that the system contains. Example 4.6 shows output from the `isalist` command on an UltraSPARC III-based system. It shows that there is a SPARC processor in the system, and that this can handle SPARC V7, V8, and V9 binaries. The processor can also handle the VIS 2.0 instruction set extensions.

**Example 4.6** Sample Output from the `isalist` Command

```
$ isalist
sparcv9+vis2 sparcv9+vis sparcv9 sparcv8plus+vis sparcv8plus sparcv8
sparcv8-fsmuld sparcv7 sparc
```

### 4.2.6 Reporting TLB Page Sizes Supported by Hardware (`pagesize`)

In Section 1.9.2 of Chapter 1 we discussed the Translation Lookaside Buffer (TLB), which the processor uses to map virtual memory addresses to physical memory addresses. Different processors are able to support different page sizes. The advantage of larger page sizes is that they let the TLB map more physical memory using a fixed number of TLB entries. For example, a TLB with 64 entries can map  $8\text{KB} \times 64 = 512\text{KB}$  when each entry is an 8KB page, but can map  $4\text{MB} \times 64 = 256\text{MB}$  when each entry holds a 4MB page. The number of different page sizes that can be supported simultaneously is hardware-dependent. Even if the hardware supports large page sizes, there is no guarantee that an application will receive large pages if it requests them. The number of available large pages depends on the amount of memory in the system and the degree to which contiguous memory is available.

The `pagesize` command prints out the different TLB page sizes that the processor can support. If no flags are specified, the utility will print the default page size. If the flag `-a` is used, it will report all the available page sizes (see Example 4.7).

### Example 4.7 Sample Output from the `pagesize` Command

```
$ pagesize -a
8192
65536
524288
4194304
```

The `pmap` command (covered in Section 4.4.7) reports the page sizes that an application has been allocated.

It is possible to change the page sizes that an application requests. You can do this in several ways.

- At compile time, you can use the `-xpagesize` compiler flag documented in Section 5.8.6 of Chapter 5.
- You can preload the Multiple PageSize Selection (`mpss.so.1`) library, which uses environment variables to set the page sizes. We will cover preloading in more detail in Section 7.2.10 of Chapter 7. An example of using preloading to set the page size for an application appears in Example 4.8. In this example, the environment is being set up to request 4MB pages for both the application stack and the heap.

### Example 4.8 Using `mpss.so.1` to Set the Page Size for an Application

```
$ setenv MPSSHEAP 4M
$ setenv MPSSSTACK 4M
$ setenv LD_PRELOAD mpss.so.1
$ a.out
```

- You can set the preferred page size for a command or for an already running application through the `ppgsz` utility. This utility takes a set of page sizes plus either a command to be run with those page sizes, or a `pid` for those page sizes to be applied to. Example 4.9 shows examples of using the `ppgsz` command.

### Example 4.9 Using the `ppgsz` Command

```
% ppgsz -o heap=4M a.out
% ppgsz -o heap=64K -p <pid>
```

Table 4.1 shows the supported page sizes for various processors.

**Table 4.1** Page Sizes Supported by Various Processor Types

Processor	4KB	8KB	64KB	512KB	2MB	4MB	32MB	256MB
UltraSPARC IIIcu		✓	✓	✓		✓		
UltraSPARC IV		✓	✓	✓		✓		
UltraSPARC IV+		✓	✓	✓		✓	✓	✓
UltraSPARC T1		✓	✓			✓		✓
UltraSPARC T2		✓	✓			✓		✓
SPARC64 VI		✓	✓	✓		✓	✓	✓
x64	✓				✓			

### 4.2.7 Reporting a Summary of SPARC Hardware Characteristics (fpversion)

`fpversion` is a tool that ships with the SPARC compiler and is not available on x86 architectures. The tool will output a summary of the processor's capabilities. The most important part of the output from `fpversion` is that it displays the options the compiler will use when it is told to optimize for the native platform (see `-xtarget=native` in Section 5.6.4 of Chapter 5).

Example 4.10 shows output from `fpversion` from an UltraSPARC IIIcu-based system.

#### Example 4.10 Output from `fpversion` on an UltraSPARC IIIcu Based System

```
$ fpversion
A SPARC-based CPU is available.
Kernel says CPU's clock rate is 1050.0 MHz.
Kernel says main memory's clock rate is 150.0 MHz.

Sun-4 floating-point controller version 0 found.
An UltraSPARC chip is available.

Use "-xtarget=ultra3cu -xcache=64/32/4:8192/512/2" code-generation option.

Hostid = 0x83xxxxxx.
```

## 4.3 Tools That Report Current System Status

### 4.3.1 Introduction

This section covers tools that report system-wide information, such as what processes are being run and how much the disk is being utilized.

### 4.3.2 Reporting Virtual Memory Utilization (vmstat)

vmstat is a very useful tool that ships with Solaris and reports the system's virtual memory and processor utilization. The information is aggregated over all the tasks of all the users of the system. Example 4.11 shows sample output from vmstat.

**Example 4.11** Sample Output from vmstat

```
$ vmstat 1
procs          memory          page          disk          faults          cpu
r  b  w  swap  free  re mf pi po fr de sr f0 sd sd --  in  sy  cs us sy id
0  0  0  5798208 1784568 25 61 1  1  1  0  0  0  1  0  0  120 170  94  9  6 85
0  0  0  5684752 1720704 0 15  0  0  0  0  0  0  0  0  0  155  35 135 50  0 50
0  0  0  5684752 1720688 0  0  0  0  0  0  0  0  0  0  0  117  10  98 50  0 50
0  0  0  5684560 1720496 0 493 0  0  0  0  0  0  0  0  0  114 260  91 49  1 50
0  0  0  5680816 1716744 2  2  0  0  0  0  0  0  0  0  0  118 196 103 50  0 50
0  0  0  5680816 1716648 18 18 0  0  0  0  0  0  0  0  0  148  23 116 50  0 50
0  0  0  5680816 1716584 0  0  0  0  0  0  0  0  0  0  0  115  19 100 50  0 50
0  0  0  5680752 1716520 0 40  0  0  0  0  0  0  22  0  0  129  14  99 50  4 46
0  0  0  5680496 1716264 0  0  0  0  0  0  0  0  0  0  0  109  24 100 50  0 50
0  0  0  5680496 1716184 11 11 0  0  0  0  0  0  0  0  0  140  23 107 50  0 50
```

Each column of the output shown in Example 4.11 represents a different metric; the command-line argument of 1 requested that vmstat report status at one-second intervals. The first row is the average of the machine since it was switched on; subsequent rows are the results at one-second intervals.

The columns that vmstat reports are as follows.

- **procs:** The first three columns report the status of processes on the system. The *r* column lists the number of processes in the run queue (i.e., waiting for CPU resources to run on), the *b* column lists the number of processes blocked (e.g., waiting on I/O, or waiting for memory to be paged in from disk), and the *w* column lists the number of processes swapped out to disk. If the number of processes in the run queue is greater than the number of virtual processors, the system may have too many active tasks or too few CPUs.
- **memory:** The two columns referring to memory show the amount of swap space available and the amount of memory on the *free* list, both reported in kilobytes. The swap space corresponds to how much data the processor can map before it runs out of virtual memory to hold it. The free list corresponds to how much data can fit into physical memory at one time. A low value for remaining swap space may cause processes to report out-of-memory errors. You can get additional information about the available swap space through the `swap` command (covered in Section 4.3.3).

- **page:** The columns labeled `re` to `sr` refer to paging information. The `re` column lists the number of pages containing data from files, either executables or data, that have been accessed again and therefore reclaimed from the list of free pages. The `mf` column lists the number of minor page faults, in which a page was mapped into the process that needed it. The `pi` column lists the number of kilobytes paged in from disk and the `po` column lists the number of kilobytes paged out to disk. The `de` column lists the anticipated short-term memory shortfall in kilobytes, which gives the page scanner a target number of pages to free up. The `sr` column lists the number of pages scanned per second. A high scan rate (`sr`) is also an indication of low memory, and that the machine is having to search through memory to find pages to send to disk. The solution is to either run fewer applications or put more memory into the machine. Continuously high values of `pi` and `po` indicate significant disk activity, due to either a high volume of I/O or to paging of data to and from disk when the system runs low on memory.
- **disk:** There is space to report on up to four disk drives, and these columns show the number of disk operations per second for each of the four drives.
- **faults:** There are three columns on faults (i.e., traps and interrupts). The `in` column lists the number of interrupts; these are used for tasks such as handling a packet of data from the network interface card. The `sy` column lists the number of system calls; these are calls into the kernel for the system to perform a task. The `cs` column lists the number of context switches, whereby one thread leaves the CPU and another is placed on the CPU.
- **cpu:** The final three columns are the percentage of user, system, and idle time. This is an aggregate over all the processors. Example 4.11 shows output from a two-CPU machine. With an idle time of 50%, this can mean that both CPUs are busy, but each only half the time, or that only one of the two CPUs is busy. In an ideal world, most of the time should be spent in user code, performing computations, rather than in the system, managing resources. Of course, this does not mean that the time in user code is being spent efficiently, just that the time isn't spent in the kernel or being idle. High levels of system time mean something is wrong, or the application is making many system calls. Investigating the cause of high system time is always worthwhile.

### 4.3.3 Reporting Swap File Usage (`swap`)

Swap space is disk space reserved for anonymous data (data that is not otherwise held on a filesystem). You can use the `swap` command to add and delete swap space from a system. It can also list the locations of swap space using the `-l` flag, and

report a summary of swap space usage under the `-s` flag. Examples of output from both of these flags is shown in Example 4.12.

#### Example 4.12 Output from the `swap` Command

```
% swap -l
swapfile          dev  swaplo   blocks    free
/dev/dsk/clt0d0s1 118,33    16 25175408 25175408
% swap -s
total: 2062392k bytes allocated + 1655952k reserved = 3718344k used, 36500448k avail-
able
```

### 4.3.4 Reporting Process Resource Utilization (`prstat`)

`prstat` was a very useful addition to Solaris 8. It prints out a list of the processes that are consuming the most processor time, which can be helpful in identifying processes that are consuming excessive amounts of CPU resources. It also reports useful values, such as the amount of memory used.

Example 4.13 shows the first few lines of output from `prstat`. It reports a screen of information, each line representing a particular process. By default, the processes are listed starting with the one that is consuming the most CPU time.

#### Example 4.13 Sample Output from `prstat`

```
PID USERNAME  SIZE  RSS STATE  PRI NICE   TIME  CPU PROCESS/NLWP
29013 martin    4904K 1944K cpu0   40  0   0:01:15 44% myapplication/1
  210 root     4504K 2008K sleep  59  0   0:27:34 0.1% automountd/2
29029 martin    4544K 4256K cpu1   59  0   0:00:00 0.1% prstat/1
  261 root     2072K   0K sleep  59  0   0:00:00 0.0% smcboot/1
...
```

The columns are as follows.

- **PID:** The process ID (PID), which is a unique number assigned to identify a particular process.
- **USERNAME:** The ID of the user owning the process.
- **SIZE:** The total size of the process. This is a measure of how much virtual address space has been allocated to the process. It does not measure how much physical memory the process is currently using.
- **RSS:** The resident set size (RSS) of the process, that is, how much of the process is actually in memory. The RSS of an application can fluctuate depending on how much data the application is currently using, and how much of the application has been swapped out to disk.

- **STATE:** The state of the process, that is, whether it is sleeping, on a CPU (as the two processes for “martin” are in the example), or waiting for a processor to run on.
- **PRI:** The priority of the process, which is a measure of how important it is for CPU time to be allocated to a particular process. The higher the priority, the more time the kernel will allow the process to be on a CPU.
- **NICE:** The nice value for the process, which allows the user to reduce the priority of an application to allow other applications to run. The higher the nice value, the less CPU time will be allocated to it.
- **TIME:** The CPU time that the process has accumulated since it started.
- **CPU:** The percentage of the CPU that the process has recently consumed.
- **PROCESS/NLWP:** The name of the executable, together with the number of lightweight processes (LWPs) in the process. From Solaris 9 onward, LWPs are equivalent to threads. `prstat` can also report activity on a per-thread basis using the `-L` flag.

You can obtain a more accurate view of system utilization by using the `prstat` command with the `-m` flag. This flag reports processor utilization using microstate accounting information. Microstate accounting is a more accurate breakdown of where the process spends its time. Solaris 10 collects microstate accounting data by default. Example 4.14 shows example output from this command.

#### Example 4.14 Output from `prstat -m`

```

PID USERNAME  USR SYS TRP TFL DFL LCK SLP LAT VCX ICX SCL SIG PROCESS/NLWP
1946 martin    0.1 0.3 0.0 0.0 0.0 0.0 100 0.0 23 0 280 0 prstat/1
5063 martin    0.2 0.0 0.0 0.0 0.0 0.0 100 0.0 24 0 95 0 gnome-panel/1
5065 martin    0.2 0.0 0.0 0.0 0.0 0.0 100 0.0 13 0 22 0 nautilus/3
7743 martin    0.1 0.0 0.0 0.0 0.0 0.0 100 0.0 61 0 76 0 soffice1.bin/6
5202 martin    0.0 0.0 0.0 0.0 0.0 0.0 100 0.0 24 2 40 0 gnome-termin/1
...
Total: 115 processes, 207 lwps, load averages: 0.00, 0.01, 0.02

```

The columns in Example 4.14 are as follows.

- **PID:** The PID of the process.
- **USERNAME:** The User ID of the process owner.
- **USR to LAT:** The percentage of time spent by the process in the various modes: user mode (USR), system mode (SYS), system traps (TRP), text (i.e., program instruction) page faults (TFL), data page faults (DFL), user locks (LCK), sleeping (SLP), and waiting for the CPU (LAT).

- **VCX and ICX:** The number of context switches, voluntary (VCX) and involuntary (ICX). A voluntary context switch is one in which the process either completes its task and yields the CPU, or enters a wait state (such as waiting for data from disk). An involuntary context switch is one in which another higher-priority task is assigned to the CPU, or the process uses up its allocation of time on the CPU.
- **SCL:** The number of system calls.
- **SIG:** The number of signals received.
- **PROCESS/NLWP:** The name of the process (PROCESS) and the number of LWPs (NLWP).

It is possible to use the `<column>` flag `-s` to sort by a particular column. In Example 4.15, this is used to sort the processes by RSS.

#### Example 4.15 Output from `prstat` Sorted by RSS

```
$ prstat -s rss
  PID USERNAME  SIZE  RSS STATE  PRI NICE   TIME  CPU PROCESS/NLWP
  8453  root        403M 222M sleep  49  0 13:17:50 0.0% Xsun/1
 28059  robin       218M 133M sleep  49  0  0:06:04 0.1% soffice2.bin/5
 28182  robin       193M  88M sleep  49  0  0:00:54 0.0% soffice1.bin/7
 26704  robin        87M  72M sleep  49  0  0:06:35 0.0% firefox-bin/4
  ...
```

### 4.3.5 Listing Processes (`ps`)

`ps` displays a list of all the processes in the system. It is a very flexible tool and has many options. The output in Example 4.16 shows one example of what `ps` can report.

#### Example 4.16 Sample Output from `ps`

```
$ ps -ef
  UID  PID  PPID  C   STIME TTY   TIME CMD
  root    0    0  0   Jul 06 ?    0:00 sched
  root    1    0  0   Jul 06 ?    0:01 /etc/init -
  root    2    0  0   Jul 06 ?    0:13 pageout
  ...
```

The options passed to the `ps` command in Example 4.16 are `-e`, to list all the processes; and `-f`, to give a “full” listing, which is a particular set of columns (in particular, it gives more information about how the application was invoked than the alternative `-l` “long” listing).

The columns in the output are as follows.

- **UID:** The UID of the user who owns the process. A large number of processes are going to be owned by root.
- **PID:** The PID of the process.
- **PPID:** The PID of the parent process.
- **C:** This column is obsolete. It used to report processor utilization used in scheduling.
- **STIME:** The start date/time of the application.
- **TTY:** The controlling terminal for the process (where the commands that go to the process are being typed). A question mark indicates that the process does not have a controlling terminal.
- **TIME:** The accumulated CPU time of the process.
- **CMD:** The command being executed (truncated to 80 characters). Under the `-f` flag, the arguments are printed as well, which can be useful for distinguishing between two processes of the same name.

One of the most useful columns is the total accumulated CPU time for a process, which is the amount of time it has been running on a CPU since it started. This column is worth watching to check that the critical programs are not being starved of time by the noncritical programs.

Most of the time it is best to pipe the output of `ps` to some other utility (e.g., `grep`), because even on an idle system there can be many processes.

### 4.3.6 Locating the Process ID of an Application (`pgrep`)

It is often necessary to find out the PID of a process to examine the process further. It is possible to do this using the `ps` command, but it is often more convenient to use the `pgrep` command. This command returns processes with names that match a given text string, or processes that are owned by a given user. Example 4.17 shows two examples of the use of this command. The first example shows the tool being used to match the name of an executable. In the example, the `-l` flag specifies that the long output format should be generated, which includes the name of the program. The second example shows the `-U` flag, which takes a username and returns a list of processes owned by that particular user—in this case, the processes owned by `root`.

### Example 4.17 Output from `pgrep`

```
% pgrep -l soff
28059 soffice2.bin
28182 soffice1.bin
% pgrep -lU root
 0 sched
 1 init
 2 pageout
 3 fsflush
760 sac
...
```

### 4.3.7 Reporting Activity for All Processors (`mpstat`)

The `mpstat` tool reports activity on a per-processor basis. It reports a number of useful measures of activity that may indicate issues at the system level. Like `vmstat`, `mpstat` takes an interval parameter that specifies how frequently the data should be reported. The first lines of output reported give the data since boot time; the rates are reported in events per second. Sample output from `mpstat` is shown in Example 4.18.

### Example 4.18 Sample Output from `mpstat`

```
$ mpstat 1
CPU minf mjf xcal  intr ithr  csw icsw migr smtx  srw syscl  usr sys  wt idl
 0  29  1  38   214  108  288  10  6  14  0  562  36  2  0  62
 1  27  1  27   44  29  177  9  6  67  0  516  33  2  0  65
CPU minf mjf xcal  intr ithr  csw icsw migr smtx  srw syscl  usr sys  wt idl
 0  7  0  11   207  103  64  9  10  0  0  7  39  1  0  60
 1  0  0  4   14  2  54  11  11  0  0  5  61  0  0  39
CPU minf mjf xcal  intr ithr  csw icsw migr smtx  srw syscl  usr sys  wt idl
 0  0  0  6   208  106  60  7  8  0  0  14  47  0  0  53
 1  0  0  65  16  9  46  6  7  0  0  4  53  2  0  45
CPU minf mjf xcal  intr ithr  csw icsw migr smtx  srw syscl  usr sys  wt idl
 0  0  0  6   204  103  36  6  3  0  0  5  68  0  0  32
 1  0  0  1   9  2  64  6  7  0  0  5  32  0  0  68
CPU minf mjf xcal  intr ithr  csw icsw migr smtx  srw syscl  usr sys  wt idl
 0  0  0  2   205  104  14  10  2  0  0  4  98  0  0  2
 1  0  0  1   34  31  93  2  2  0  0  15  2  0  0  98
CPU minf mjf xcal  intr ithr  csw icsw migr smtx  srw syscl  usr sys  wt idl
 0  0  0  8   204  104  40  2  6  0  0  5  51  0  0  49
 1  0  0  0   11  2  58  8  6  0  0  5  49  0  0  51
....
```

Each line of output corresponds to a particular CPU for the previous second. The columns in the `mpstat` output are as follows.

- CPU: The ID of the CPU to which the data belongs. Data is reported on a per-CPU basis.

- `minf`: The number of minor page faults per second. These occur when a page of memory is mapped into a process.
- `mjf`: The number of major page faults per second. These occur when the requested page of data has to be brought in from disk.
- `xcal`: The number of interprocess cross-calls per second. This occurs when a process on one CPU requests action from another. An example of this is where memory is unmapped through a call to `munmap`. The `munmap` call will use a cross call to ensure that other CPUs also remove the mapping to the target memory range from their TLB.
- `intr`: The number of interrupts per second.
- `ithr`: The number of interrupt threads per second, not counting the clock interrupt. These are lower-priority interrupts that are handled by threads that are scheduled onto the processor to handle the interrupt event.
- `csw`: The number of context switches per second, where the process either voluntarily yields its time on the processor before the end of its allocated slot or is involuntarily displaced by a higher-priority process.
- `icsw`: The number of involuntary context switches per second, where the process is removed from the processor either to make way for a higher-priority thread or because it has fully utilized its time slot.
- `migr`: The number of thread migrations to another processor per second. Usually, best performance is obtained if the operating system keeps the process on the same CPU. In some instances, this may not be possible and the process is migrated to a different CPU.
- `smtx`: The number of times a mutex lock was not acquired on the first try.
- `srw`: The number of times a read/write lock was not acquired on the first try.
- `syscl`: The number of system calls per second.
- `usr`: The percentage of time spent in user code.
- `sys`: The percentage of time spent in system code.
- `wt`: The percentage of time spent waiting on I/O. From Solaris 10 onward, this will report zero because the method of calculating wait time has changed.
- `idl`: The percentage of time spent idle.

In the code in Example 4.18, the two processors are spending about 50% of their time in user code and 50% of their time idle. In fact, just a single process is running. What is interesting is that this process is migrating between the two processors (you can see this in the migrations per second). It is also apparent that processor 0 is handling most of the interrupts.

### 4.3.8 Reporting Kernel Statistics (`kstat`)

`kstat` is a very powerful tool for returning information about the kernel. The counts it produces are the number of events since the processor was switched on. So, to determine the number of events that an application causes, it is necessary to run `kstat` before and after the application, and determine the difference between the two values. Of course, this is accurate only if just one process is running on the system. Otherwise, the other processes can change the numbers.

One of the metrics that `kstat` reports is the number of emulated floating-point instructions. Not all floating-point operations are performed in hardware; some have been left to software. Obviously, software is more flexible, but it is slower than hardware, so determining whether an application is doing any floating-point operations in software can be useful. An example of checking for unfinished floating-point traps is shown in Example 4.19. The `-p` option tells `kstat` to report statistics in a parsable format; the `-s` option selects the statistic of interest.

#### Example 4.19 Using `kstat` to Check for Unfinished Floating-Point Traps

```
$ kstat -p -s 'fpu_unfinished_traps'
unix:0:fpu_traps:fpu_unfinished_traps      32044940
$ a.out
$ kstat -p -s 'fpu_unfinished_traps'
unix:0:fpu_traps:fpu_unfinished_traps      32044991
```

Example 4.19 shows the number of unfinished floating-point operations reported by `kstat` before the application ran, and the number afterward. The difference between the two values is 51, which means that 51 unfinished floating-point operations were handled by trapping to software between the two calls to `kstat`. It is likely that these traps were caused by the application `a.out`, but if there was other activity on the system, these traps cannot be confidently attributed to any one particular process. To have some degree of confidence in the number of traps on a busy system, it is best to repeat the measurement several times, and to measure the number of traps that occur when the application is not running.

### 4.3.9 Generating a Report of System Activity (`sar`)

The `sar` utility records system activity over a period of time into an archive for later analysis. It is possible to select which aspects of system performance are recorded. Once an archive of data has been recorded, `sar` is also used to extract the particular activities of interest.

To record a `sar` data file it is necessary to specify which system events should be recorded, the name of the file in which to record the events, the interval

between samples, and the number of samples you want. An example command line for `sar` is shown in Example 4.20.

#### Example 4.20 Example Command Line for `sar`

```
$ sar -A -o /tmp/sar.dat 5 10
```

This instructs `sar` to do the following.

1. Record all types of events (-A).
2. Store the events in the file `/tmp/sar.dat`.
3. Record a sample at 5-second intervals.
4. Record a total of 10 samples.

When `sar` runs it will output data to the screen as well as to the data file, as shown in Example 4.21.

#### Example 4.21 Output from `sar` as It Runs

```
$ sar -A -o /tmp/sar.dat 5 10

SunOS machinename 5.9 Generic_112233-01 sun4u      08/26/2003

21:07:39   %usr  %sys   %wio   %idle
           device %busy  avque  r+w/s  blks/s  await  avserv
           runq-sz %runocc swpq-sz %swpocc
           bread/s lread/s %rcache bwrit/s lwrit/s %wcache pread/s      pwrite/s
           swpin/s bswin/s swpot/s bswot/s pswch/s
           scall/s sread/s swrit/s fork/s  exec/s  rchar/s wchar/s
           iget/s namei/s dirbk/s
           rawch/s canch/s outch/s rcvin/s xmtin/s mdmin/s
           proc-sz ov      inod-sz ov      file-sz ov      lock-sz
           msg/s  sema/s
           atch/s pgin/s  ppgin/s  pflt/s  vflt/s  slock/s
           pgout/s ppgout/s pgfree/s pgscan/s %ufs_ipf
           freemem freeswap
           sml_mem alloc   fail    lg_mem  alloc   fail    ovsz_alloc fail

21:07:44      50      1      0      50
           fd0      0      0.0    0      0      0.0    0.0
           ssd0     0      0.0    0      0      0.0    0.0
           ssd0,a   0      0.0    0      0      0.0    0.0
           ssd0,b   0      0.0    0      0      0.0    0.0
           ssd0,c   0      0.0    0      0      0.0    0.0
           ssd0,h   0      0.0    0      0      0.0    0.0
           ssd1     0      0.0    0      0      0.0    0.0
           ssd1,a   0      0.0    0      0      0.0    0.0
```

*continues*

**Example 4.21** Output from `sar` as It Runs (*continued*)

```

    ssd1,b      0          0.0    0    0    0.0    0.0
    ssd1,c      0          0.0    0    0    0.0    0.0
    ssd1,h      0          0.0    0    0    0.0    0.0
    0.0         0          0.0    0    0    0.0    0.0
    0           0          100    0    0    100    0    0
    0.00        0.0       0.00   0.0  99    0.00   0.00
    76          6          14    0.00 0.00  1550   2850
    0           0          0
    0           0          161    0    0    0
    65/30000    0 157574/157574 0    0/0  0
    0.00        0.00
    0.00        0.00      0.00  0.60 2.20  0.00
    0.00        0.00      0.00  0.00 0.00
    247682 17041644
    0         0          0    0    0    0 17858560  0

```

Example 4.21 presents a lot of information. The text at the beginning supplies a template that indicates what the counters represent. The information is as follows.

- First it reports the time the system spent in user (`%usr`), system (`%sys`), waiting for block I/O (`%wio`), and idle (`%idle`).
- Next is a section for each device that reports the device name, percentage of time busy (`%busy`), average queue length while the device was busy (`avque`), number of reads and writes per second (`r+w/s`), number of 512-byte blocks transferred per second (`blk/s`), average wait time in ms (`avwait`), and average service time in ms (`avserv`).
- The length of the queue of runnable processes (`runq_sz`) and the percentage of time occupied (`%runocc`) are listed next. The fields `swpq-sz` and `%swpocc` no longer have values reported for them.
- Next is the number of transfers per second of buffers to disk or other block devices. Read transfers per second (`bread/s`), reads of system buffers (`lread/s`), cache hit rate for reads (`%rcache`), write transfers per second (`bwrit/s`), writes of system buffers (`lwrit/s`), cache hit rate for writes (`%wcache`), raw physical device reads (`pread/s`), and raw physical device writes (`pwrit/s`) are included.
- Swapping activity is recorded as the number of swap-ins per second (`swpin/s`), number of blocks of 512 bytes swapped in (`bswin/s`), number of swap-outs per second (`swpot/s`), number of 512-byte blocks swapped out (`bswot/s`), and number of process switches per second (`pswch/s`). The number of 512-byte blocks transferred includes the loading of programs.
- System calls are reported as the total number of system calls per second (`scall/s`), number of read calls per second (`sread/s`), number of write calls

per second (`swrit/s`), number of forks per second (`fork/s`), number of execs per second (`exec/s`), number of characters transferred by read (`rchar/s`), and number of characters transferred by write (`wchar/s`).

- Next is a report of file access system routines called per second. The number of files located by inode entry (`iget/s`), number of file system pathname searches (`namei/s`), and number of directory block reads (`dirbk/s`) are included.
- TTY I/O reports stats on character I/O to the controlling terminal. This includes raw character rate (`rawch/s`), processed character rate (`canch/s`), output character rate (`outch/s`), receive rate (`rcvin/s`), transmit rate (`xmtin/s`), and modem interrupts per second (`mdmin/s`).
- Process, inode, file, and lock table sizes are reported as `proc-sz`, `inod-sz`, `file-sz`, and `lock_sz`. The associated overflow (`ov`) fields report the overflows that occur between samples for each table.
- The number of messages and semaphores per second is reported as `msg/s` and `sema/s`.
- Paging to memory is reported as the number of page faults per second that were satisfied by reclaiming a page from memory (`atch/s`), the number of page-in requests per second (`pgin/s`), the number of page-ins per second (`ppgin/s`), the number of copy on write page faults per second (`pflt/s`), the number of page not in memory faults per second (`vflt/s`), and the number of faults per second caused by software locks requiring physical I/O (`slock/s`).
- Paging to disk is reported as the number of requested page-outs per second (`pgout/s`), the number of page-outs per second (`ppgout/s`), the number of pages placed on the free list per second (`pgfree/s`), the number of pages scanned per second (`pgscan/s`), and the percentage of igets that required a page flush (`%ufs_ipf`).
- Free memory is reported as the average number of pages available to user processes (`freemem`), and the number of disk blocks available for swapping (`freeswap`).
- Kernel memory allocation is reported as a small memory pool of free memory (`sml_mem`), the number of bytes allocated from the small memory pool (`alloc`), and the number of requests for small memory that failed (`fail`). Similar counters exist for the large pool (`lg_mem`, `alloc`, `fail`). The amount of memory allocated for oversize requests is reported as `ovsz_alloc`, and the number of times this failed as `fail`.

The command to read an existing `sar` output file is shown in Example 4.22.

#### Example 4.22 Command Line to Instruct `sar` to Read an Existing Data File

```
$ sar -A -f /tmp/sar.dat
```

This asks `sar` to output all (-A) the information from the `sar` archive (/tmp/sar.dat). It is possible to request that `sar` output only a subset of counters.

In the `sar` output shown in Example 4.21, it is apparent that the CPU is 50% busy (in fact, it is a two-CPU system, and one CPU is busy compiling an application), and that there is some character output and some read and write system calls. It is reasonably apparent that the system is CPU-bound, although it has additional CPU resources which could potentially be used to do more work.

### 4.3.10 Reporting I/O Activity (`iostat`)

The `iostat` utility is very similar to `vmstat`, except that it reports I/O activity rather than memory statistics.

The first line of output from `iostat` is the activity since boot. Subsequent lines represent the activity over the time interval between reports. Example output from `iostat` is shown in Example 4.23.

#### Example 4.23 Example of `iostat` Output

```
% iostat 1
  tty          ssd0          ssd1          nfs1          nfs58          cpu
tin tout kps tps serv kps tps serv kps tps serv kps tps serv us sy wt id
0      2  17  1  90  22  1  45  0  0  0  0  0  27  20  1  0  79
0 234  0  0  0  8  1  6  0  0  0  0  0  0  50  2  0  48
0  80  0  0  0  0  0  0  0  0  0  0  0  0  50  0  0  50
0  80  0  0  0  560  4  16  0  0  0  0  0  0  46  2  1  50
0  80  0  0  0  352  4  13  0  0  0  0  0  0  48  8  0  44
0  80  0  0  0  560  15  13  0  0  0  0  0  0  42  6  2  50
```

The information is as follows.

- The first two columns give the number of characters read (`tin`) and written (`tout`) for the `tty` devices.
- The next four sets of three columns give information for four disks. The `kps` column lists the number of kilobytes per second, `tps` the number of transfers per second, and `serv` the average service time in ms.
- CPU time is reported as a percentage in user (`us`), system (`sy`), waiting for I/O (`wt`), and idle (`id`).

Another view of I/O statistics is provided by passing the `-Cnx` option to `iostat`. Output from this is shown in Example 4.24.

**Example 4.24** Output from `iostat -Cnx 1`

```
$ iostat -Cnx 1
....
          extended device statistics
  r/s    w/s    kr/s    kw/s  wait  actv  wsvc_t  asvc_t  %w    %b  device
  0.0    25.0    0.0    594.0  0.0  0.3   0.0    12.7   0     4   c0
  0.0     0.0    0.0     0.0  0.0  0.0   0.0     0.0   0     0   c1
  0.0     0.0    0.0     0.0  0.0  0.0   0.0     0.0   0     0   c0t0d0
  0.0    25.0    0.0    594.0  0.0  0.3   0.0    12.7   0     8   c0t1d0
  0.0     0.0    0.0     0.0  0.0  0.0   0.0     0.0   0     0   c1t2d0
  ...
```

In Example 4.24, each disk gets a separate row in the output. The output comprises the following columns:

- `r/s`: Reads per second
- `w/s`: Writes per second
- `kr/s`: Kilobytes read per second
- `kw/s`: Kilobytes written per second
- `wait`: Average number of transactions waiting for service
- `actv`: Average number of transactions actively being serviced
- `wsvc_t`: Average service time in wait queue, in milliseconds
- `asvc_t`: Average time actively being serviced, in milliseconds
- `%w`: Percentage of time the waiting queue is nonempty
- `%b`: Percentage of time the disk is busy
- `device`: The device that this applies to

In the example shown in Example 4.24, the busy device is `c0t1d0`, which is writing out about 600KB/s from 25 writes (about 24KB/write), each write taking about 13 ms. The device is busy about 8% of the time and has an average of about 0.3 writes going on at any one time.

If a disk is continuously busy more than about 20% of the time, it is worth checking the average service time, or the time spent waiting in the queue, to ensure that these are low. Once the disk starts to become busy, the service times may increase significantly. If this is the case, it may be worth investigating whether to spread file activity over multiple disks. The `iostat` options `-e` and `-E` will report the errors that have occurred for each device since boot.

### 4.3.11 Reporting Network Activity (`netstat`)

`netstat` can provide a variety of different reports. The `-s` flag shows statistics per protocol. A sample of the output showing the statistics for the IPv4 protocol is shown in Example 4.25.

**Example 4.25** Example of `netstat -s` Output

```
% netstat -s
...
IPv4  ipForwarding      = 2      ipDefaultTTL      = 255
      ipInReceives   =8332869 ipInHdrErrors      = 0
      ipInAddrErrors = 0      ipInCksumErrs     = 0
      ipForwDatagrams = 0      ipForwProhibits   = 0
      ipInUnknownProtos = 2      ipInDiscards      = 0
      ipInDelivers   =8316558 ipOutRequests      =13089344
      ipOutDiscards  = 0      ipOutNoRoutes     = 0
      ipReasmTimeout = 60      ipReasmReqds      = 0
      ipReasmOKs     = 0      ipReasmFails      = 0
      ipReasmDuplicates = 0      ipReasmPartDups   = 0
      ipFragOKs      = 0      ipFragFails       = 0
      ipFragCreates  = 0      ipRoutingDiscards = 0
      tcpInErrs      = 0      udpNoPorts        = 17125
      udpInCksumErrs = 0      udpInOverflows    = 0
      rawipInOverflows = 0      ipsecInSucceeded  = 0
      ipsecInFailed  = 0      ipInIPv6          = 0
      ipOutIPv6      = 0      ipOutSwitchIPv6   = 213
...

```

You can obtain a report showing input and output from `netstat -i`; an example is shown in Example 4.26. This output shows the number of packets sent and received, the number of errors, and finally the number of collisions.

**Example 4.26** Example of `netstat -i` Output

```
% netstat -i 1
input  eri0  output          input (Total)  output
packets errs  packets errs  colls  packets errs  packets errs  colls
486408174 5  499073054 3  0  530744745 5  543409625 3  0
      5 0  9 0  0  12 0  16 0  0
      6 0  10 0  0  13 0  17 0  0
      6 0  10 0  0  14 0  18 0  0

```

The collision rate is the number of collisions divided by the number of output packets. A value greater than about 5% to 10% may indicate a problem. Similarly, you can calculate error rates by dividing the number of errors by the total input or output packets. An error rate greater than about one-fourth of a percent may indicate a problem.

### 4.3.12 The `snoop` command

The `snoop` command, which you must run with superuser privileges, gathers information on the packets that are passed over the network. It is a very powerful way of examining what the network is doing, and consequently the command has a large number of options. In “promiscuous” mode, it gathers all packets that the local machine can see. In nonpromiscuous mode (enabled using the `-P` flag), it only gathers information on packages that are addressed to the local machine. It is also possible to gather a trace of the packets (using the `-o` flag) for later processing by the `snoop` command (using the `-i` flag). The packets collected (or examined) can be filtered in various ways, perhaps most usefully by the machines communicating, alternatively individual packets can be printed out. An example of the output from the `snoop` command is shown in Example 4.27.

**Example 4.27** Output from the `snoop` Command

```
$ snoop
Using device /dev/eri (promiscuous mode)
here -> mc1 TCP D=1460 S=5901 Ack=2068723218 Seq=3477475694 Len=0 Win=50400
here -> mc2 TCP D=2049 S=809 Ack=3715747853 Seq=3916150345 Len=0 Win=49640
mc1 -> here TCP D=22 S=1451 Ack=3432082168 Seq=2253017191 Len=0 Win=33078
...
```

Note that `snoop` can capture and display unencrypted data being passed over the network. As such, use of this tool may have privacy, policy, or legal issues in some domains.

### 4.3.13 Reporting Disk Space Utilization (`df`)

The `df` command reports the amount of space used on the disk drives. Example output from the `df` command is shown in Example 4.28.

**Example 4.28** Example Output from the `df` Command

```
% df -k1
Filesystem          kbytes    used  avail capacity  Mounted on
/dev/dsk/c0t1d0s0   3096423  1172450 1862045    39%    /
/proc                0          0        0      0%    /proc
mnttab               0          0        0      0%    /etc/mnttab
fd                   0          0        0      0%    /dev/fd
swap                 9475568     48  9475520     1%    /var/run
swap                 9738072   262552  9475520     3%    /tmp
/dev/dsk/c0t1d0s7   28358357  26823065 1251709    96%    /data
/dev/dsk/c0t2d0s7   28814842  23970250 4556444    85%    /export/home
```

The `-k1` option tells `df` to report disk space in kilobytes (rather than as the number of 512-byte blocks), and to only report data for local drives. The columns are reasonably self-explanatory and include the name of the disk, the size, the amount used, the amount remaining, and the percentage amount used. The final column shows the mount point. In this example, both the `/data` and the `/export/home` file systems are running low on available space. On Solaris 9 and later there is a `-h` option to produce the output in a more human-readable format.

### 4.3.14 Reporting Disk Space Used by Files (`du`)

The `du` utility reports the disk space used by a given directory and its subdirectories. Once again, there is a `-k` option to report usage in kilobytes. On Solaris 9 and later, there is also a `-h` option to report in a human-readable format. Example output from the `du` command is shown in Example 4.29.

**Example 4.29** Example of Output from the `du` Command

```
% du -k
8  ./X11-unix
8  ./X11-pipe
3704 .
% du -h
8K  ./X11-unix
8K  ./X11-pipe
3.6M .
```

The `du` command in Example 4.29 reported that two directories consume 8KB each, and there is about 3.6MB of other data in the current directory.

## 4.4 Process- and Processor-Specific Tools

### 4.4.1 Introduction

This section covers tools that report the status of a particular process, or the events encountered by a particular processor.

### 4.4.2 Timing Process Execution (`time`, `timex`, and `ptime`)

The commands `time`, `timex`, and `ptime` all report the amount of time that a process uses. They all have the same syntax, as shown in Example 4.30. All three tools produce output showing the time a process spends in user code and system code, as well as reporting the elapsed time, or wall time, for the process. The wall

time is the time between when the process started and when it completed. A multi-threaded process will typically report a combined user and system time that is greater than the wall time. The tools do differ in output format. The tool `time` can be passed additional options that will cause it to report more information.

### Example 4.30 Syntax of the `time` Command

```
% time <app> <params>
```

## 4.4.3 Reporting System-Wide Hardware Counter Activity (`cpustat`)

`cpustat` first shipped with Solaris 8. It is a tool for inspecting the hardware performance counters on all the processors in a system. The performance counters report events that occur to the processor. For example, one counter may be incremented every time data is fetched from memory, and another counter may be incremented every time an instruction is executed. The events that can be counted, and the number of events that can be counted simultaneously, are hardware-dependent. Opteron processors can typically count four different event types at the same time, whereas UltraSPARC processors typically only count two. We will discuss hardware performance counters in greater depth in Chapter 10.

`cpustat` reports the number of performance counter events on a system-wide basis, hence it requires superuser permissions to run. So, if multiple programs are running, the reported values will represent the events encountered by all programs. If the system is running a mix of workloads, this information may not be of great value, but if the system is performing a single job, it is quite possible that this level of aggregation of data will provide useful information.

Assume that the system is dedicated to a single task—the program of interest—and the program is in some kind of steady state (e.g., it is a Web server that is dealing with many incoming requests). The command line for `cpustat`, shown in Example 4.31, is appropriate for an UltraSPARC IIICu-based system. The output is a way of determining which performance counters are worth further investigation.

### Example 4.31 Sample Command Line for `cpustat` to Collect System-Wide Stats

```
$ cpustat -c pic0=Dispatch0_IC_miss,pic1=Dispatch0_mispred,sys \
-c pic0=Rstall_storeQ,pic1=Re_DC_miss,sys \
-c pic0=EC_rd_miss,pic1=Re_EC_miss,sys \
-c pic0=Rstall_IU_use,pic1=Rstall_FP_use,sys \
-c pic0=Cycle_cnt,pic1=Re_PC_miss,sys \
-c pic0=Instr_cnt,pic1=DTLB_miss,sys \
-c pic0=Cycle_cnt,pic1=Re_RAW_miss,sys
```

When the `-c` flag is passed to `cpustat` (and `cputrack`) it provides a pair of counters on which to collect. These are referred to as `pic0` and `pic1`. More than 60 event types are available to select from on the UltraSPARC IIIc processor, and two can be selected at once. Some of the event types are available on only one of the counters, so not every pairing is possible. The `,sys` appended at the end of the pair of counter descriptions indicates that the counters should also be collected during system time. The counters are collected in rotation, so each pair of counters is collected for a short period of time. The default interval is five seconds.

If the program is not in a steady state—suppose it reads some data from memory and then spends the next few seconds in intensive floating-point operations—it is quite possible that the coarse sampling used earlier will miss the interesting points (e.g., looking for cache misses during the floating-point-intensive code, and looking for floating-point operations when the data is being fetched from memory). Example 4.32 shows the command line for `cputrack` to rotate through a selection of performance counters, and partial output from the command.

#### Example 4.32 Example of `cpustat` Output

```
$ cpustat      -c pic0=Rstall_storeQ,pic1=Re_DC_miss,sys \
>              -c pic0=EC_rd_miss,pic1=Re_EC_miss,sys \
>              -c pic0=Rstall_IU_use,pic1=Rstall_FP_use,sys \
>              -c pic0=Cycle_cnt,pic1=Re_PC_miss,sys \
>              -c pic0=Instr_cnt,pic1=DTLB_miss,sys \
>              -c pic0=Cycle_cnt,pic1=Re_RAW_miss,sys
time cpu event  pic0      pic1
5.005 0  tick 294199 1036736 # pic0=Rstall_storeQ,pic1=Re_DC_miss,sys
5.005 1  tick 163596 12604317 # pic0=Rstall_storeQ,pic1=Re_DC_miss,sys
10.005 0 tick   5485  965974 # pic0=EC_rd_miss,pic1=Re_EC_miss,sys
10.005 1 tick  76669 11598139 # pic0=EC_rd_miss,pic1=Re_EC_miss,sys
...
```

The columns of `cpustat` output shown in Example 4.32 are as follows.

- The first column reports the time of the sample. In this example, the samples are being taken every five seconds.
- The next column lists the CPU identifier. The samples are taken and reported for each CPU.
- The next column lists the type of event. For `cpustat`, the type of event is only going to be a `tick`.
- The next two columns list the counts for performance counters `pic0` and `pic1` since the last tick event.
- Finally, if `cpustat` is rotating through counters, the names of the counters are reported after the `#` sign.

#### 4.4.4 Reporting Hardware Performance Counter Activity for a Single Process (`cpustrack`)

`cpustrack` first shipped with Solaris 8. It is another tool that reports the number of performance counter events. However, `cpustrack` has the advantages of collecting events only for the process of interest and reporting the total number of such events at the end of the run. This makes it very useful for situations in which the application starts, does something, and then exits.

The script in Example 4.33 shows one way that `cpustrack` might be invoked on a process.

##### Example 4.33 Script for Invoking `cpustrack` on an Application

```
$ cpustrack -c pic0=Dispatch0_IC_miss,pic1=Dispatch0_mispred,sys \
-c pic0=Rstall_storeQ,pic1=Re_DC_miss,sys \
-c pic0=EC_rd_miss,pic1=Re_EC_miss,sys \
-c pic0=Rstall_IU_use,pic1=Rstall_FP_use,sys \
-c pic0=Cycle_cnt,pic1=Re_PC_miss,sys \
-c pic0=Instr_cnt,pic1=DTLB_miss,sys \
-c pic0=Cycle_cnt,pic1=Re_RAW_miss,sys \
-o <desired location of results file> \
<application>
```

The script in Example 4.33 demonstrates how to use `cpustrack` to rotate through the counters and capture data about the run of an application. The same caveat applies as for `cpustat`: Rotating through counters may miss the events of interest. An alternative way to invoke `cpustrack` is to give it just a single pair of counters. The example in Example 4.34 shows this.

##### Example 4.34 Example of `cpustrack` on a Single Pair of Counters

```
$ cpustrack -c pic0=Cycle_cnt,pic1=Re_DC_miss testcode
time lwp      event      pic0      pic1
1.118  1         tick      663243149 14353162
2.128  1         tick      899742583  9706444
3.118  1         tick      885525398  7786122
3.440  1         exit      2735203660 33964190
```

The output in Example 4.34 shows a short program that runs for three seconds. `cpustrack` has counted the number of processor cycles consumed by the application using counter 0, and the number of data-cache miss events using counter 1; both numbers are per second, except for the line marked “exit,” which contains the total counts over the entire run. The columns in the output are as follows.

- `time`: The time at which the sample was taken. In this case, the samples were taken at one-second intervals.
- `lwp`: The LWP that is being sampled. If the `-f` option is passed to `cputrack`, it will follow child processes. In this mode, data from other LWPs will be interleaved.
- `event`: The event type, such as `ticks` or the `exit` line. Each tick event is the number of events since the last tick. The `exit` line occurs when a process exits, and it reports the total number of events that occurred over the duration of the run. The event line also reports data at points where the process forks or joins.
- `pic0` and `pic1`: The last two columns report the number of events for the two performance counters. If `cputrack` were rotating through performance counters, the names of the performance counters would be reported after a `#` sign.

It is also possible to attach `cputrack` to a running process. The option for this is `-p <pid_id>`, and `cputrack` will report the events for that process.

#### 4.4.5 Reporting Bus Activity (`busstat`)

The `busstat` tool reports performance counter events for the system bus. The available performance counters are system-dependent. The `-l` option lists the devices that have performance counter statistics available. The `-e` option will query what events are available on a particular device.

The currently set performance counters can be read using the `-r` option. To select particular performance counters it is necessary to use the `-w` option, but this requires superuser privileges. An example of using `busstat` to measure memory activity on an UltraSPARC T1-based system is shown in Example 4.35.

##### Example 4.35 Using `busstat` to Query Memory Activity on an UltraSPARC T1

```
# busstat -l
Busstat Device(s):
dram0 dram1 dram2 dram3 jbus0
# busstat -e dram0
pic0
mem_reads
mem_writes
....
pic1
mem_reads
mem_writes
...
# busstat -w dram0,pic0=mem_reads,pic1=mem_writes
time dev      event0          pic0      event1          pic1
1   dram0  mem_reads      45697     mem_writes      8775
2   dram0  mem_reads      37827     mem_writes      3767
```

### 4.4.6 Reporting on Trap Activity (`trapstat`)

`trapstat` is a SPARC-only tool that first shipped with Solaris 9 and enables you to look at the number of traps the kernel is handling. It counts the number of traps on a system-wide basis, hence it requires superuser privileges. For example, it is a very useful tool for looking at TLB misses on the UltraSPARC II. Example 4.36 shows output from `trapstat`.

**Example 4.36** Sample Output from `trapstat`

vct name	cpu0
20 fp-disabled	6
24 cleanwin	31
41 level-1	4
44 level-4	0
46 level-6	2
49 level-9	1
4a level-10	100
4e level-14	101
60 int-vec	3
64 itlb-miss	3
68 dtlb-miss	144621
6c dtlb-prot	0
84 spill-user-32	0
8c spill-user-32-cln	0
98 spill-kern-64	612
a4 spill-asuser-32	0
ac spill-asuser-32-cln	199
c4 fill-user-32	0
cc fill-user-32-cln	70
d8 fill-kern-64	604
108 syscall-32	26

In the output shown in Example 4.36, a number of data TLB traps are occurring. This is the number per second, and 144,000 per second is not an unusually high number. Each trap takes perhaps 100 ns, so this corresponds to a few milliseconds of real time. We discussed TLB misses in greater detail in Section 4.2.6.

A high rate of any trap is a cause for concern. The traps that most often have high rates are TLB misses (either instruction [`itlb-miss`] or data [`dtlb-miss`]), and spill and fill traps.

Spill and fill traps indicate that the code is making many calls to routines, and each call may make further subcalls before returning (think of the program having to run up and then down a flight of stairs for each function call and its corresponding return). Each time the processor makes a call, it needs a fresh register window. When no register window is available, the processor will trap so that the operating system can provide one. I discussed register windows in Section 2.3.3 of Chapter 2. It may be possible to avoid this by either compiling with crossfile optimizations enabled (as discussed in Section 5.7.2 of Chapter 5), or restructuring the code so that each call will do more work.

It is possible to make `trapstat` run on a single process. The command line for this is shown in Example 4.37.

#### Example 4.37 Command Line to Run `trapstat` on a Single Program

```
# trapstat <program> <args>
```

At the end of the run of the process, this will report the number of traps that the single process caused. The figures will be reported (by default) as the number of traps per second.

### 4.4.7 Reporting Virtual Memory Mapping Information for a Process (`pmap`)

The `pmap` utility returns information about the address space of a given process. Possibly the most useful information the utility returns is information about the page size mapping returned under the `-s` option.

Example 4.38 shows a sample of output from the `pmap` utility under the `-s` option. The output is useful in that it shows where the code and data are located in memory, as well as where the libraries are located in memory. For each memory range, a page size is listed in the `Pgsz` column. In this case, all the memory has been allocated on 8KB pages (I discussed page sizes in Section 1.9.2 of Chapter 1). Output from `pmap` is the best way to determine whether an application has successfully obtained large pages.

#### Example 4.38 `pmap -s` Output

```
% pmap -s 7962
7962: ./myapp params
Address Kbytes Pgsz Mode Mapped File
00010000 272K 8K r-x-- /export/home/myapp
00054000 80K - r-x-- /export/home/myapp
00068000 32K 8K r-x-- /export/home/myapp
0007E000 48K 8K rwx-- /export/home/myapp
...
000D2000 2952K 8K rwx-- [ heap ]
...
004D4000 1984K 8K rwx-- [ heap ]
006C4000 8K - rwx-- [ heap ]
006C6000 50944K 8K rwx-- [ heap ]
...
FF210000 8K 8K r-x-- /usr/platform/sun4u-us3/lib/libc_psr.so.1
FF220000 32K 8K r-x-- /opt/SUNWspro/prod/usr/lib/libCrun.so.1
...
```

The page size for an application can be controlled at runtime (see Section 4.2.6) or at compile time (see Section 5.8.6 of Chapter 5).

### 4.4.8 Examining Command-Line Arguments Passed to Process (`pargs`)

The `pargs` command reports the arguments passed to a process. The command takes either a `pid` or a core file. An example of this command is shown in Example 4.39.

**Example 4.39** Example of `pargs`

```
$ sleep 60&
[1] 18267
$ pargs 18267
18267: sleep 60
argv[0]: sleep
argv[1]: 60
```

### 4.4.9 Reporting the Files Held Open by a Process (`pfiles`)

The `pfiles` utility reports the files that a given `pid` has currently opened, together with information about the file's attributes. An example of the output from this command is shown in Example 4.40.

**Example 4.40** Output from `pfiles`

```
% pfiles 7093
7093: -csh
Current rlimit: 256 file descriptors
0: S_IFCHR mode:0666 dev:118,32 ino:3422 uid:0 gid:3 rdev:13,2
   O_RDONLY|O_LARGEFILE
1: S_IFCHR mode:0666 dev:118,32 ino:3422 uid:0 gid:3 rdev:13,2
   O_RDONLY|O_LARGEFILE
2: S_IFCHR mode:0666 dev:118,32 ino:3422 uid:0 gid:3 rdev:13,2
   O_RDONLY|O_LARGEFILE
...
```

### 4.4.10 Examining the Current Stack of Process (`pstack`)

The `pstack` tool prints out the stack dump from either a running process or a core file. An example of using this tool to print the stack of the `sleep` command is shown in Example 4.41. This tool is often useful in determining whether an application is still doing useful computation or whether it has hit a point where it is making no further progress.

### Example 4.41 Output from `pstack`

```
% sleep 10 &
[1] 4556
% pstack 4556
4556:  sleep 10
      ff31f448 sigsuspend (ffbffaa8)
      00010a28 main      (2, ffbffbd8, ffbffbe8, 20c00, 0, 0) + 120
      000108f0 _start   (0, 0, 0, 0, 0, 0) + 108
```

## 4.4.11 Tracing Application Execution (`truss`)

`truss` is a useful utility for looking at the calls from an application to the operating system, calls to libraries, or even calls within an application. Example 4.42 shows an example of running the application `ls` under the `truss` command.

### Example 4.42 Output of the `truss` Command Running on `ls`

```
$ truss ls
execve("/usr/bin/ls", 0xFFBFFBE4, 0xFFBFFBEC)  argc = 1
mmap(0x00000000, 8192, PROT_READ|PROT_WRITE|PROT_EXEC,
      MAP_PRIVATE|MAP_ANON, -1, 0) = 0xFF3B0000
resolvepath("/usr/bin/ls", "/usr/bin/ls", 1023) = 11
resolvepath("/usr/lib/ld.so.1", "/usr/lib/ld.so.1", 1023) = 16
stat("/usr/bin/ls", 0xFFBFF9C8)              = 0
open("/var/ld/ld.config", O_RDONLY)          Err#2 ENOENT
open("/usr/lib/libc.so.1", O_RDONLY)         = 3
fstat(3, 0xFFBFF304)                         = 0
....
```

When an application is run under `truss` the tool reports every call the operating system made. This can be very useful when trying to determine what an application is doing. The `-f` flag will cause `truss` to follow forked processes.

When `truss` is run with the `-c` flag it will provide a count of the number of calls made, as well as the total time accounted for by these calls. Example 4.43 shows the same `ls` command run, but this time only the count of the number of calls is collected.

### Example 4.43 Call Count for the `ls` Command Using `truss`

```
$ truss -c ls
syscall      seconds  calls  errors
_exit        .000    1
write        .000    35
open         .000    7      3
close        .000    4
time         .000    1
brk          .000    10
```

**Example 4.43** Call Count for the `ls` Command Using `truss` (*continued*)

```

stat                .000      1
fstat               .000      3
ioctl               .000      3
execve              .000      1
fcntl               .000      1
mmap                .000      7
munmap              .000      1
memcntl             .000      1
resolvepath         .000      5
getdents64          .000      3
lstat64             .000      1
fstat64             .000      2
-----
sys totals:         .002      87      3
usr time:           .001
elapsed:           .020

```

It is also possible to run `truss` on an existing process. This will generate the same output as invoking `truss` on the process initially. This is useful for checking whether a process is still doing something. Example 4.44 shows the command line to do this.

**Example 4.44** Attaching `truss` to an Existing Process

```
$ truss -p <pid>
```

It is also possible to use `truss` to print out the calls within an application. The `-u` flag takes the names of the libraries of interest, or `a.out` to represent the application. The tool will then report calls made by these modules, as shown in Example 4.45.

**Example 4.45** `truss` Used to Show Calls Made within an Application

```

% truss -u a.out bzip2
execve("bzip2", 0xFFBFFB84, 0xFFBFFB8C)  argc = 1
-> atexit(0x2be04, 0x44800, 0x0, 0x0)
-> mutex_lock(0x456e8, 0x0, 0x0, 0x0)
-> _return_zero(0x456e8, 0x0, 0x0, 0x0)
<- mutex_lock() = 0
-> mutex_unlock(0x456e8, 0x1, 0x0, 0x0)
-> _return_zero(0x456e8, 0x1, 0x0, 0x0)
<- mutex_unlock() = 0
<- atexit() = 0
-> _init(0x0, 0x44800, 0x0, 0x0)
-> _check_threaded(0x44a68, 0x0, 0x0, 0x0)
-> thr_main(0x0, 0x0, 0x0, 0x0)
-> _return_negone(0x0, 0x0, 0x0, 0x0)
<- thr_main() = -1
<- _check_threaded() = 0x44a68
<- _init() = 0
-> main(0x1, 0xffbffb84, 0xffbffb8c, 0x44800)
-> signal(0x2, 0x1724c, 0x0, 0x0)
....

```

### 4.4.12 Exploring User Code and Kernel Activity with `dtrace`

The `dtrace` utility is part of Solaris 10, and it offers an unprecedented view into the behavior of user code and system code. Using `dtrace`, it is possible to count the number of times a routine gets called, time how long a routine takes, examine the parameters that are passed into a routine, and even find out what a routine was doing when a given event happened. All of this makes it a very powerful tool, but to describe it in sufficient detail is beyond the scope of this book. A simple example of the use of `dtrace` is shown in Example 4.46. This script counts the number of calls to `malloc`, and for each call it records the size of the memory requested.

#### Example 4.46 `dtrace` Script to Count the Calls to `malloc`

```
#!/usr/sbin/dtrace -s
pid$target:libc.so:malloc:entry
{
    @proc[probefunc]=count();
    @array[probefunc]=quantize(arg0);
}

END
{
    printa(@proc);
    printa(@array);
}
```

The output from this script, when run on the command `ls`, is shown in Example 4.47. The output shows that `malloc` is called 15 times, and displays a histogram of the requested memory sizes.

#### Example 4.47 Number and Size of Calls to `malloc` by `ls`

```
# ./malloc.d -c ls
dtrace: script './malloc.d' matched 2 probes
...
dtrace: pid 3645 has exited
CPU    ID          FUNCTION:NAME
16     2              :END
malloc                                15

malloc
value  ----- Distribution ----- count
   4   |                                0
   8   | @@@@@                            2
  16   | @@@@@@@@@@@@@@@@@@@@           6
  32   | @@@                              1
  64   | @@@@@@@@@@                     3
 128   |                                0
```

**Example 4.47** Number and Size of Calls to `malloc` by `ls` (*continued*)

256		2
512		0
1024		0
2048		0
4096		0
8192		0
16384		0
32768		1
65536		0

Once the sizes of the memory requests have been determined, it may also be interesting to find out where the memory requests are being made. The script shown in Example 4.48 captures the call stack for the calls to `malloc`.

**Example 4.48** Capturing the User Call Stack for Calls to `malloc`

```
#!/usr/sbin/dtrace -s
pid$target:libc.so:malloc:entry
{
    @stack[ustack(5)]=count();
    @array[probefunc]=quantize(arg0);
}

END
{
    printa(@stack);
    printa(@array);
}
```

An example of running the script to capture the call stack of calls to `malloc` is shown in Example 4.49.

**Example 4.49** Call Stack for `malloc` Requests Made by the Compiler

```
# ./malloc_stack.d -c "cc -O an.c"
dtrace: script './malloc_stack.d' matched 2 probes
dtrace: pid 4063 has exited
CPU    ID          FUNCTION:NAME
  6     2                      ;END
...
      cc`malloc
      libc.so.1`calloc+0x58
      cc`stralloc+0x8
      cc`str_newcopy+0xc
      cc`addopt+0x88
      154
```

*continues*

**Example 4.49** Call Stack for `malloc` Requests Made by the Compiler (*continued*)

```

malloc
value  ----- Distribution ----- count
   0 |                                     0
   1 |                                     2
   2 | @@@                                 21
   4 | @@@@@@                               52
   8 | @@@@@@@@@@                          88
  16 | @@@@@@@@@@                          65
  32 | @@@@@@                               52
  64 | @@@@@@                               44
 128 | @                                     5
 256 |                                     2
 512 |                                     1
1024 |                                     0

```

## 4.5 Information about Applications

### 4.5.1 Reporting Library Linkage (`ldd`)

The `ldd` utility reports the shared libraries that are linked into an application. This is useful for acquiring information, but it should not have any effect on performance (unless the wrong version of a library is selected somehow).

The output in Example 4.50 shows the library the application is searching for on the left, and the library that has been located on the right.

**Example 4.50** Output from `ldd` Showing the Linking of a Particular Application

```

$ ldd ap27
libm.so.1 => /usr/lib/libm.so.1
libc.so.1 => /usr/lib/libc.so.1
/usr/platform/SUNW,Sun-Blade-2500/lib/libc_psr.so.1

```

Passing the `-r` option to `ldd` will cause it to check both the objects that are linked into the application and the particular function calls that are required. It will report whether the application is missing a library, and it will report the functions that are missing.

The output shown in Example 4.51 is from the `-r` option passed to `ldd`. There are two items of interest. First, `ldd` reports that it is unable to locate the `libsunmath` library, which is Sun's library of additional mathematical functions. Under this option, `ldd` reports the two function calls that it is unable to locate, and these function calls correspond to square root calls for single-precision floating-point, and for long integers.

**Example 4.51** The `-r` Option for `ldd`

```

$ ldd -r someapp
libdl.so.1 => /usr/lib/libdl.so.1
libnsl.so.1 => /usr/lib/libnsl.so.1
libgen.so.1 => /usr/lib/libgen.so.1
libm.so.1 => /usr/lib/libm.so.1
libc.so.1 => /usr/lib/libc.so.1
libsocket.so.1 => /usr/lib/libsocket.so.1
libsunmath.so.1 => (file not found)
libelf.so.1 => /usr/lib/libelf.so.1
libmp.so.2 => /usr/lib/libmp.so.2
/usr/platform/SUNW,Sun-Blade-1000/lib/libc_psr.so.1
symbol not found: sqrtf
(someapp)
symbol not found: sqrtl
(someapp)

```

The paths where libraries are located are hard-coded into the application at link time. I will cover the procedure for doing this in more detail in Section 7.2.6 of Chapter 7. At runtime, it is possible to use the `LD_LIBRARY_PATH` environment variable to override where the application finds libraries, or to assist the application in locating a particular library. So, for the case in Example 4.51, if the `LD_LIBRARY_PATH` variable were set to point to a directory containing `libsunmath.so`, `ldd` would report that the application used that version of the library. Example 4.52 shows an example of setting the `LD_LIBRARY_PATH` environment variable under `csh`. Of course, you can use the same environment variable to change where the application loads all its libraries from, so be careful when setting it and do not rely on it as the default mechanism to enable an application locating its libraries at deployment.

**Example 4.52** Example of Setting the `LD_LIBRARY_PATH` Variable

```

$ setenv LD_LIBRARY_PATH /export/home/my_libraries/

```

The `LD_LIBRARY_PATH` environment variable will override the search path for both 32-bit and 64-bit applications. To explicitly set search paths for these two application types you can use the environment variables `LD_LIBRARY_PATH_32` and `LD_LIBRARY_PATH_64`.

It is also possible to set the `LD_PRELOAD` environment variable to specify a library that is to be loaded before the application. This enables the use of a different library in addition to the one shipped with the application. This can be a useful way to debug the application's interactions with libraries. I will cover this in more detail in Section 7.2.10 of Chapter 7.

The `-u` option will request that `ldd` report any libraries that are linked to the application but not used. In Example 4.53, both `libm` (the math library) and `libsocket` (the sockets library) are linked into the application but not actually used.

### Example 4.53 Example of `ldd -u` to Check for Unused Libraries

```
$ ldd -u ./myapp
libdl.so.1 => /usr/lib/libdl.so.1
libnsl.so.1 => /usr/lib/libnsl.so.1
libm.so.1 => /usr/lib/libm.so.1
libc.so.1 => /usr/lib/libc.so.1
libsocket.so.1 => /usr/lib/libsocket.so.1
/usr/platform/SUNW,Sun-Blade-1000/lib/libc_psr.so.1

unused object=/usr/lib/libm.so.1
unused object=/usr/lib/libsocket.so.1
```

Another useful option for `ldd` is the `-i` flag. This requests that `ldd` report the order in which the libraries will be initialized. The output from `ldd` shown in Example 4.54 indicates that `libc` is initialized first, and `libsocket` is initialized last.

### Example 4.54 Example of `ldd -i` Output

```
$ ldd -i ./thisapp
libdl.so.1 => /usr/lib/libdl.so.1
libnsl.so.1 => /usr/lib/libnsl.so.1
libm.so.1 => /usr/lib/libm.so.1
libc.so.1 => /usr/lib/libc.so.1
libsocket.so.1 => /usr/lib/libsocket.so.1
libmp.so.2 => /usr/lib/libmp.so.2
/usr/platform/SUNW,Sun-Blade-1000/lib/libc_psr.so.1

init object=/usr/lib/libc.so.1
init object=/usr/lib/libmp.so.2
init object=/usr/lib/libnsl.so.1
init object=/usr/lib/libsocket.so.1
```

## 4.5.2 Reporting the Type of Contents Held in a File (`file`)

The `file` tool reports on the type of a particular file. It can be useful for situations when it is necessary to check whether a particular application is a script wrapper for the actual real application, or the real application. Another way this tool can help is in determining on what type of processor a given application will run. Recall that the `isalist` tool from Section 4.2.5 reported the processor's architecture; the `file` tool will report the architecture an application requires. For a

given application to run on a particular machine, the processor needs to support the application's architecture.

Example 4.55 shows `file` being run on an application. The binary is 32-bit and requires at least a v8plus architecture to run.

#### Example 4.55 Example of Running `file` on an Application

```
$ file a.out
a.out:          ELF 32-bit MSB executable SPARC32PLUS Version 1, V8+ Required, dynami-
cally linked, not stripped
```

The `file` command is often useful when examining files or libraries to determine why linking failed with an error reporting an attempt to link 32-bit and 64-bit objects.

### 4.5.3 Reporting Symbols in a File (`nm`)

The `nm` tool reports the symbols defined inside a library, object file, or executable. Typically, this tool will dump out a lot of information. The useful information is usually the names of routines defined in the file, and the names of routines the file requires. If the file has been stripped (using the `strip` utility), no information is reported. A snippet of example output from `nm` is shown in Example 4.56.

#### Example 4.56 Short Sample of Output from `nm`

```
$ nm a.out
a.out:
[Index]  Value      Size  Type  Bind  Other  Shndx  Name
[45]    | 133144 |      4 | OBJT | WEAK | 0      | 15    | environ
[60]    | 132880 |      0 | FUNC | GLOB | 0      | UNDEF | exit
[37]    | 67164  |     40 | FUNC | LOCL | 0      | 8     | foo
[53]    | 67204  |     48 | FUNC | GLOB | 0      | 8     | main
[42]    | 132904 |      0 | FUNC | GLOB | 0      | UNDEF | printf
...
```

The output from `nm` shown in Example 4.56 indicates that `a.out` defines a couple of routines, such as `main` and `foo`, but depends on libraries to provide the routines `exit` and `printf`.

### 4.5.4 Reporting Library Version Information (`pvcs`)

It is possible to define multiple versions of a library in a single library file. This is an important mechanism to allow older applications to run with newer versions

of a library. The older library API is still available, and the older applications will link to these versions. The newer API is also present, and the newer applications will link to this.

The `pvs` utility prints out information about the functions and versions of those functions that a library exports, or the library versions that a library or executable requires. By default, `pvs` will report both the definitions in the library and the requirements of the library.

Example 4.57 shows `pvs` reporting the versions of the libraries that the `ls` executable requires.

#### Example 4.57 Libraries Required by the `ls` Command

```
% pvs /bin/ls
    libc.so.1 (SUNW_1.19, SUNWprivate_1.1);
```

The `-r` option, for displaying only the requirements of the file, can be used to show that `libc.so.1` requires `libdl.so.1`, as demonstrated in Example 4.58.

#### Example 4.58 Requirements of `libc.so.1`

```
% pvs -r /usr/lib/libc.so.1
    libdl.so.1 (SUNW_1.4, SUNWprivate_1.1);
```

The `-d` option shows the versions defined in the library. Example 4.59 shows part of the output of the versions defined in `libc.so.1`.

#### Example 4.59 Versions Defined in `libc.so.1`

```
% pvs -d /usr/lib/libc.so.1
    libc.so.1;
    SUNW_1.21.2;
    SUNW_1.21.1;
    SUNW_1.21;
    SUNW_1.20.4;
    ....
```

It is also possible to list the symbols defined in a library using the `-s` flag. Part of the output of this for `libdl.so.1` is shown in Example 4.60.

**Example 4.60** Versions of Functions Exported by `libdl.so.1`

```
$ pvs -ds /usr/lib/libdl.so.1
libdl.so.1:
    _DYNAMIC;
    _edata;
    _etext;
    _end;
    _PROCEDURE_LINKAGE_TABLE_;
SUNW_1.4:
    dladdr1;
SUNW_1.3:
SUNW_1.2:
SUNW_1.1:
    dlmopen;
    dldump;
    dlinfo;
...
```

### 4.5.5 Examining the Disassembly of an Application, Library, or Object (`dis`)

The `dis` utility will disassemble libraries, applications, and object files. An example of this is shown in Example 4.61.

**Example 4.61** Example of Using `dis`

```
$ /usr/ccs/bin/dis a.out
**** DISASSEMBLER ****

disassembly for a.out

section .text
_start()
    10694: bc 10 20 00      clr        %fp
    10698: e0 03 a0 40      ld        [%sp + 0x40], %l0
    1069c: 13 00 00 83      sethi    %hi(0x20c00), %o1
    106a0: e0 22 61 8c      st        %l0, [%o1 + 0x18c]
    106a4: a2 03 a0 44      add      %sp, 0x44, %l1
...
```

### 4.5.6 Reporting the Size of the Various Segments in an Application, Library, or Object (`size`)

The `size` utility prints the size in bytes of the various segments in an application, library, or object file. When used without parameters the command reports the size of the `text` (executable code), `data` (initialized data), and `bss` (uninitialized data). The `-f` flag reports the name of each allocatable segment together with its size in bytes. The `-n` flag also reports the nonloadable segments (these segments contain metadata such as debug information). An example is shown in Example 4.62.

#### Example 4.62 Using the `size` Command

```
% size a.out
3104 + 360 + 8 = 3472
% size -fn a.out
17(.interp) + 304(.hash) + 592(.dynsym) + 423(.dynstr) + 48(.SUNW_version) +
12(.rela.data) + 72(.rela.plt) + 1388(.text) + 16(.init) + 12(.fini) + 4(.rodata) +
4(.got) + 124(.plt) + 184(.dynamic) + 48(.data) + 8(.bss) + 1152(.symtab) +
525(.strtab) + 248(.debug_info) + 53(.debug_line) + 26(.debug_abbrev) + 650(.comment) +
184(.shstrtab) = 6094
```

### 4.5.7 Reporting Metadata Held in a File (`dumpstabs`, `dwarfdump`, `elfdump`, `dump`, and `mcs`)

It is possible to extract information about how an application was built using the `dumpstabs` utility, which is shipped with the compiler. This utility reports a lot of information, but the most useful is the command line that was passed to the compiler. Two other utilities serve a similar purpose: `dwarfdump`, which reports the data for applications built with the dwarf debug format, and `elfdump` which reports similar information for object files. All three utilities can take various flags to specify the level of detail, but by default, `dumpstabs` and `elfdump` print out all information, whereas `dwarfdump` does not report anything for versions earlier than Sun Studio 11 (in these cases, use the `-a` flag to print all the information). Applications built with the Sun Studio 10 compiler (and earlier) default to the `stabs` format, so `dumpstabs` is the appropriate command to use. In Sun Studio 11, the C compiler switched to using dwarf format. In Sun Studio 12, all the compilers default to using dwarf format.

Example 4.63 shows an example of building a file using Sun Studio 10, and then using `dumpstabs` and `grep` to extract the compile line used to build the file. In general, a lot of information is reported by `dumpstabs`, so passing the output through `grep` and searching for either the name of the file or the `CMDLINE` marker will reduce the output substantially.

### Example 4.63 Example of Searching for the Command Line for a Compiler Using Sun Studio 10

```
$ cc -fast -o test test.c
$ dumpstabs test | grep test.c
36:          test.c 00000000 00000000 LOCAL FILE ABS
0:  .stabs "test.c",N_UNDF,0x0,0x3,0xb8
2:  .stabs "/export/home; /opt/SUNWspro/prod/bin/cc -fast -c test.c",N_CMD-
LINE,0x0,0x0,0x0
```

A similar set of actions for Sun Studio 11 and `dwarfdump` is shown in Example 4.64.

### Example 4.64 Example of Searching for the Command Line for a Compiler Using Sun Studio 11

```
$ cc -fast -o test test.c
$ dwarfdump test | grep command_line
      DW_AT_SUN_command_line  /opt/SUNWspro/prod/bin/cc -fast -c test.c
< 13> DW_AT_SUN_command_line  DW_FORM_string
```

It is also possible to use the `dump` command with the `-sv` option to extract most of the information from an executable. This will dump all the sections in an executable, printing those that are text in text format and the other sections as hexadecimal. An example of the output from `dump` is shown in Example 4.65. The actual output from the command runs to a number of pages, and Example 4.65 shows only a small part of this output.

### Example 4.65 Example of Output from `dump`

```
$ dump -sv a.out

a.out:
.interp:
      2f 75 73 72 2f 6c 69 62 2f 6c 64 2e 73 6f 2e 31 00

.hash:
      00 00 00 95 00 00 00 8e 00 00 00 00 00 00 00 00

....
**** STRING TABLE INFORMATION ****
```

*continues*

**Example 4.65** Example of Output from `dump` (*continued*)

```
.strtab:
  <offset>      Name
  <0>
  <1>           a.out
  <7>           crt1.s
  <14>          crt1.s
  <21>          __get_exit_frame_monitor_ptr
...
.stab.indexstr:
  <offset>      Name
  <115>        /tmp/;/opt/SUNWspro/prod/bin/f90 -g -qoption f90comp -
h.XAzwWCA01y4\SDCK. test.f90
...
```

The `mcs` tool, which is shipped with Solaris, manipulates the comments section in `elf` files. The `-p` option will print the comments. It is possible to delete the comments section using `-d`, or append more strings using `-a`. The comments section often holds details of the compiler version used and the header files included. An example of manipulating the comments section is shown in Example 4.66. The initial comments section shows the version information for the compiler, together with details of the header files included at compile time. Using the `mcs` flag `-a`, it is possible to append another comment to the file.

**Example 4.66** Manipulating the Comments Section Using `mcs`

```
$ cc -O code.c
$ mcs -p a.out
a.out:

cg: Sun Compiler Common 11 2005/10/13
cg: Sun Compiler Common 11 2005/10/13
@(#)stdio.h 1.84 04/09/28 SMI
@(#)feature_tests.h 1.25 07/02/02 SMI
...
ld: Software Generation Utilities - Solaris Link Editors: 5.10-1.486
$ mcs -a "Hello" a.out
$ mcs -p a.out
a.out:

...
ld: Software Generation Utilities - Solaris Link Editors: 5.10-1.486
Hello
```