3

Processes

Contributions from Denis Sheahan

Monitoring process activity is a routine task during the administration of systems. Fortunately, a large number of tools examine process details, most of which make use of `procfs`. Many of these tools are suitable for troubleshooting application problems and for analyzing performance.

3.1 Tools for Process Analysis

Since there are so many tools for process analysis, it can be helpful to group them into general categories.

- **Overall status tools.** The `prstat` command immediately provides a by-process indication of CPU and memory consumption. `prstat` can also fetch microstate accounting details and by-thread details. The original command for listing process status is `ps`, the output of which can be customized.
- **Control tools.** Various commands, such as `pkill`, `pstop`, `prun` and `preap`, control the state of a process. These commands can be used to repair application issues, especially runaway processes.
- **Introspection tools.** Numerous commands, such as `pstack`, `pmap`, `pfiles`, and `pargs` inspect process details. `pmap` and `pfiles` examine the memory and file resources of a process; `pstack` can view the stack backtrace of a process and its threads, providing a glimpse of which functions are currently running.

- **Lock activity examination tools.** Excessive lock activity and contention can be identified with the `plockstat` command and DTrace.
- **Tracing tools.** Tracing system calls and function calls provides the best insight into process behavior. Solaris provides tools including `truss`, `apptrace`, and `dtrace` to trace processes.

Table 3.1 summarizes and cross-references the tools covered in this section.

Table 3.1 Tools for Process Analysis

Tool	Description	Reference
`prstat`	For viewing overall process status	3.2
`ps`	To print process status and information	3.3
`ptree`	To print a process ancestry tree	3.4
`pgrep`; `pkill`	To match a process name; to send a signal	3.4
`pstop`; `prun`	To freeze a process; to continue a process	3.4
`pwait`	To wait for a process to finish	3.4
`preap`	To reap zombies	3.4
`pstack`	For inspecting stack backtraces	3.5
`pmap`	For viewing memory segment details	3.5
`pfiles`	For listing file descriptor details	3.5
`ptime`	For timing a command	3.5
`psig`	To list signal handlers	3.5
`pldd`	To list dynamic libraries	3.5
`pflags`; `pcred`	To list tracing flags; to list process credentials	3.5
`pargs`; `pwdx`	To list arguments, env; to list working directory	3.5
`plockstat`	For observing lock activity	3.6
`truss`	For tracing system calls and signals, and tracing function calls with primitive details	3.7
`apptrace`	For tracing library calls with processed details	3.7
`dtrace`	For safely tracing any process activity, with minimal effect on the process and system	3.7

Many of these tools read statistics from the `/proc` file system, `procfs`. See Section 2.10 in *Solaris™ Internals*, which discusses `procfs` from introduction to implementation. Also refer to `/usr/include/sys/procfs.h` and the `proc(4)` man page.

3.2 Process Statistics Summary: prstat

The process statistics utility, `prstat`, shows us a top-level summary of the processes that are using system resources. The `prstat` utility summarizes this information every 5 seconds by default and reports the statistics for that period.

```
$ prstat
   PID USERNAME  SIZE   RSS STATE  PRI NICE      TIME  CPU PROCESS/NLWP
 25646 rmc      1613M   42M cpu15    0   10   0:33:10 3.1% filebench/2
 25661 rmc      1613M   42M cpu8     0   10   0:33:11 3.1% filebench/2
 25652 rmc      1613M   42M cpu20    0   10   0:33:09 3.1% filebench/2
 25647 rmc      1613M   42M cpu0     0   10   0:33:10 3.1% filebench/2
 25641 rmc      1613M   42M cpu27    0   10   0:33:10 3.1% filebench/2
 25656 rmc      1613M   42M cpu7     0   10   0:33:10 3.1% filebench/2
 25634 rmc      1613M   42M cpu11    0   10   0:33:11 3.1% filebench/2
 25637 rmc      1613M   42M cpu17    0   10   0:33:10 3.1% filebench/2
 25643 rmc      1613M   42M cpu12    0   10   0:33:10 3.1% filebench/2
 25648 rmc      1613M   42M cpu1     0   10   0:33:10 3.1% filebench/2
 25640 rmc      1613M   42M cpu26    0   10   0:33:10 3.1% filebench/2
 25651 rmc      1613M   42M cpu31    0   10   0:33:10 3.1% filebench/2
 25654 rmc      1613M   42M cpu29    0   10   0:33:10 3.1% filebench/2
 25650 rmc      1613M   42M cpu5     0   10   0:33:10 3.1% filebench/2
 25653 rmc      1613M   42M cpu10    0   10   0:33:10 3.1% filebench/2
 25638 rmc      1613M   42M cpu18    0   10   0:33:10 3.1% filebench/2
 25660 rmc      1613M   42M cpu13    0   10   0:33:10 3.1% filebench/2
 25635 rmc      1613M   42M cpu25    0   10   0:33:10 3.1% filebench/2
 25642 rmc      1613M   42M cpu28    0   10   0:33:10 3.1% filebench/2
 25649 rmc      1613M   42M cpu19    0   10   0:33:08 3.1% filebench/2
 25645 rmc      1613M   42M cpu3     0   10   0:33:10 3.1% filebench/2
 25657 rmc      1613M   42M cpu4     0   10   0:33:09 3.1% filebench/2
Total: 91 processes, 521 lwps, load averages: 29.06, 28.84, 26.68
```

The default output for `prstat` shows one line of output per process. Entries are sorted by CPU consumption. The columns are as follows:

- **PID.** The process ID of the process.
- **USERNAME.** The real user (login) name or real user ID.
- **SIZE.** The total virtual memory size of mappings within the process, including all mapped files and devices.
- **RSS.** Resident set size. The amount of physical memory mapped into the process, including that shared with other processes. See Section 6.7.
- **STATE.** The state of the process. See Chapter 3 in *Solaris™ Internals*.
- **PRI.** The priority of the process. Larger numbers mean higher priority. See Section 3.7 in *Solaris™ Internals*.
- **NICE.** Nice value used in priority computation. See Section 3.7 in *Solaris™ Internals*.

- **TIME.** The cumulative execution time for the process, printed in CPU hours, minutes, and seconds.
- **CPU.** The percentage of recent CPU time used by the process.
- **PROCESS/NLWP.** The name of the process (name of executed file) and the number of threads in the process.

3.2.1 Thread Summary: `prstat -L`

The `-L` option causes `prstat` to show one thread per line instead of one process per line.

```
$ prstat -L
   PID USERNAME  SIZE   RSS STATE  PRI NICE      TIME  CPU PROCESS/LWPID
 25689 rmc      1787M  217M sleep   59    0   0:00:08 0.1% filebench/1
 25965 rmc      1785M  214M cpu22   60   10   0:00:00 0.1% filebench/2
 26041 rmc      1785M  214M cpu4    60   10   0:00:00 0.0% filebench/2
 26016 rmc      1785M  214M sleep   60   10   0:00:00 0.0% filebench/2
     9 root       10M 9648K sleep   59    0   0:00:14 0.0% svc.configd/14
     9 root       10M 9648K sleep   59    0   0:00:26 0.0% svc.configd/12
 26174 rmc      5320K 5320K cpu30   59    0   0:00:00 0.0% prstat/1
     9 root       10M 9648K sleep   59    0   0:00:36 0.0% svc.configd/10
     7 root       19M   17M sleep   59    0   0:00:11 0.0% svc.startd/9
    93 root     2600K 1904K sleep   59    0   0:00:00 0.0% syseventd/12
    93 root     2600K 1904K sleep   59    0   0:00:00 0.0% syseventd/11
    93 root     2600K 1904K sleep   59    0   0:00:00 0.0% syseventd/10
    93 root     2600K 1904K sleep   59    0   0:00:00 0.0% syseventd/9
    93 root     2600K 1904K sleep   59    0   0:00:00 0.0% syseventd/8
    93 root     2600K 1904K sleep   59    0   0:00:00 0.0% syseventd/7
    93 root     2600K 1904K sleep   59    0   0:00:00 0.0% syseventd/6
    93 root     2600K 1904K sleep   59    0   0:00:00 0.0% syseventd/5
...
```

The output is similar to the previous example, but the last column is now represented by process name and thread number:

- **PROCESS/LWPID.** The name of the process (name of executed file) and the lwp ID of the lwp being reported.

3.2.2 Process Microstates: `prstat -m`

The process microstates can be very useful to help identify why a process or thread is performing suboptimally. By specifying the `-m` (show microstates) and `-L` (show per-thread) options, you can observe the per-thread microstates. The microstates represent a time-based summary broken into percentages of each thread. The columns `USR` through `LAT` sum to 100% of the time spent for each thread during the `prstat` sample.

```
$ prstat -mL
   PID USERNAME USR SYS TRP TFL DFL LCK SLP LAT VCX ICX SCL SIG PROCESS/LWPID
 25644 rmc       98 1.5 0.0 0.0 0.0 0.0 0.0 0.1   0  36 693   0 filebench/2
 25660 rmc       98 1.7 0.1 0.0 0.0 0.0 0.0 0.1   2  44 693   0 filebench/2
 25650 rmc       98 1.4 0.1 0.0 0.0 0.0 0.0 0.1   0  45 699   0 filebench/2
 25655 rmc       98 1.4 0.1 0.0 0.0 0.0 0.0 0.2   0  46 693   0 filebench/2
 25636 rmc       98 1.6 0.1 0.0 0.0 0.0 0.0 0.2   1  50 693   0 filebench/2
 25651 rmc       98 1.6 0.1 0.0 0.0 0.0 0.0 0.2   0  54 693   0 filebench/2
 25656 rmc       98 1.5 0.1 0.0 0.0 0.0 0.0 0.2   0  60 693   0 filebench/2
 25639 rmc       98 1.5 0.1 0.0 0.0 0.0 0.0 0.2   1  61 693   0 filebench/2
 25634 rmc       98 1.3 0.1 0.0 0.0 0.0 0.0 0.4   0  63 693   0 filebench/2
 25654 rmc       98 1.3 0.1 0.0 0.0 0.0 0.0 0.4   0  67 693   0 filebench/2
 25659 rmc       98 1.7 0.1 0.0 0.0 0.0 0.0 0.4   1  68 693   0 filebench/2
 25647 rmc       98 1.5 0.1 0.0 0.0 0.0 0.0 0.4   0  73 693   0 filebench/2
 25648 rmc       98 1.6 0.1 0.0 0.0 0.0 0.3 0.2   2  48 693   0 filebench/2
 25643 rmc       98 1.6 0.1 0.0 0.0 0.0 0.0 0.5   0  75 693   0 filebench/2
 25642 rmc       98 1.4 0.1 0.0 0.0 0.0 0.0 0.5   0  80 693   0 filebench/2
 25638 rmc       98 1.4 0.1 0.0 0.0 0.0 0.0 0.6   0  76 693   0 filebench/2
 25657 rmc       97 1.8 0.1 0.0 0.0 0.0 0.4 0.3   6  64 693   0 filebench/2
 25646 rmc       97 1.7 0.1 0.0 0.0 0.0 0.0 0.6   6  83 660   0 filebench/2
 25645 rmc       97 1.6 0.1 0.0 0.0 0.0 0.0 0.9   0  55 693   0 filebench/2
 25652 rmc       97 1.7 0.2 0.0 0.0 0.0 0.0 0.9   2 106 693   0 filebench/2
 25658 rmc       97 1.5 0.1 0.0 0.0 0.0 0.0 1.0   0  72 693   0 filebench/2
 25637 rmc       97 1.7 0.1 0.0 0.0 0.0 0.3 0.6   4  95 693   0 filebench/2
Total: 91 processes, 510 lwps, load averages: 28.94, 28.66, 24.39
```

As discussed in Section 2.11, you can use the USR and SYS states to see what percentage of the elapsed sample interval a process spent on the CPU, and LAT as the percentage of time waiting for CPU. Likewise, you can use the TFL and DTL to determine if and by how much a process is waiting for memory paging—see Section 6.6.1. The remainder of important events such as disk and network waits are bundled into the SLP state, along with other kernel wait events. While SLP column is inclusive of disk I/O, other types of blocking can cause time to be spent in the SLP state. For example, kernel locks or condition variables also accumulate time in this state.

3.2.3 Sorting by a Key: `prstat -s`

The output from prstat can be sorted by a set of keys, as directed by the -s option. For example, if we want to show processes with the largest physical memory usage, we can use prstat -s rss.

```
$ prstat -s rss
   PID USERNAME  SIZE   RSS STATE  PRI NICE      TIME  CPU PROCESS/NLWP
 20340 ftp       183M  176M sleep   59    0   0:00:24 0.0% httpd/1
  4024 daemon     11M   10M sleep   59    0   0:00:06 0.0% nfsmapid/19
  2632 daemon     11M 9980K sleep   59    0   0:00:06 0.0% nfsmapid/5
     7 root       10M 9700K sleep   59    0   0:00:05 0.0% svc.startd/14
     9 root     9888K 8880K sleep   59    0   0:00:08 0.0% svc.configd/46
 21091 ftp        13M 8224K sleep   59    0   0:00:00 0.0% httpd/1
   683 root     7996K 7096K sleep   59    0   0:00:07 0.0% svc.configd/16
   680 root     7992K 7096K sleep   59    0   0:00:07 0.0% svc.configd/15
```

continues

```
   PID USERNAME  SIZE   RSS STATE  PRI NICE      TIME  CPU PROCESS/NLWP
   671 root     7932K 7068K sleep   59    0   0:00:04 0.0% svc.startd/13
   682 root     7956K 7064K sleep   59    0   0:00:07 0.0% svc.configd/43
   668 root     7924K 7056K sleep   59    0   0:00:03 0.0% svc.startd/13
   669 root     7920K 7056K sleep   59    0   0:00:03 0.0% svc.startd/15
   685 root     7876K 6980K sleep   59    0   0:00:07 0.0% svc.configd/15
   684 root     7824K 6924K sleep   59    0   0:00:07 0.0% svc.configd/16
   670 root     7796K 6924K sleep   59    0   0:00:03 0.0% svc.startd/12
   687 root     7712K 6816K sleep   59    0   0:00:07 0.0% svc.configd/17
   664 root     7668K 6756K sleep   59    0   0:00:03 0.0% svc.startd/12
   681 root     7644K 6752K sleep   59    0   0:00:08 0.0% svc.configd/13
   686 root     7644K 6744K sleep   59    0   0:00:08 0.0% svc.configd/17
...
```

The following are valid keys for sorting:

- **cpu.** Sort by process CPU usage. This is the default.
- **pri.** Sort by process priority.
- **rss.** Sort by resident set size.
- **size.** Sort by size of process image.
- **time.** Sort by process execution time.

The `-S` option sorts by ascending order, rather than descending.

3.2.4 User Summary: `prstat -t`

A summary by user ID can be printed with the `-t` option.

```
$ prstat -t
 NPROC USERNAME  SIZE   RSS MEMORY      TIME  CPU
   233 root      797M  477M    48%   0:05:31 0.4%
    50 daemon    143M   95M   9.6%   0:00:12 0.0%
    14 40000     112M   28M   2.8%   0:00:00 0.0%
     2 rmc      9996K 3864K   0.4%   0:00:04 0.0%
     2 ftp       196M  184M    19%   0:00:24 0.0%
     2 50000    4408K 2964K   0.3%   0:00:00 0.0%
    18 nobody    104M   51M   5.2%   0:00:00 0.0%
     8 webservd   48M   21M   2.1%   0:00:00 0.0%
     7 smmsp      47M   10M   1.0%   0:00:00 0.0%
Total: 336 processes, 1201 lwps, load averages: 0.02, 0.01, 0.01
```

3.2.5 Project Summary: `prstat -J`

A summary by project ID can be generated with the `-J` option. This is very useful for summarizing per-project resource utilization. See Chapter 7 in *Solaris™ Internals* for information about using projects.

```
$ prstat -J
   PID USERNAME  SIZE   RSS STATE  PRI NICE      TIME  CPU PROCESS/NLWP
 21130 root     4100K 3264K cpu0    59    0   0:00:00 0.2% prstat/1
 21109 root     7856K 2052K sleep   59    0   0:00:00 0.0% sshd/1
 21111 root     1200K  952K sleep   59    0   0:00:00 0.0% ksh/1
  2632 daemon     11M 9980K sleep   59    0   0:00:06 0.0% nfsmapid/5
   118 root     3372K 2372K sleep   59    0   0:00:06 0.0% nscd/24

PROJID    NPROC  SIZE   RSS MEMORY      TIME  CPU PROJECT
     3        8   39M   18M   1.8%   0:00:00 0.2% default
     0      323 1387M  841M    85%   0:05:58 0.0% system
    10        3   18M 8108K   0.8%   0:00:04 0.0% group.staff
     1        2   19M 6244K   0.6%   0:00:09 0.0% user.root

Total: 336 processes, 1201 lwps, load averages: 0.02, 0.01, 0.01
```

3.2.6 Zone Summary: **prstat -Z**

The `-Z` option provides a summary per zone. See Chapter 6 in *Solaris™ Internals* for more information about Solaris Zones.

```
$ prstat -Z
   PID USERNAME  SIZE   RSS STATE  PRI NICE      TIME  CPU PROCESS/NLWP
 21132 root     2952K 2692K cpu0    49    0   0:00:00 0.1% prstat/1
 21109 root     7856K 2052K sleep   59    0   0:00:00 0.0% sshd/1
  2179 root     4952K 2480K sleep   59    0   0:00:21 0.0% automountd/3
 21111 root     1200K  952K sleep   49    0   0:00:00 0.0% ksh/1
  2236 root     4852K 2368K sleep   59    0   0:00:06 0.0% automountd/3
  2028 root     4912K 2428K sleep   59    0   0:00:10 0.0% automountd/3
   118 root     3372K 2372K sleep   59    0   0:00:06 0.0% nscd/24

ZONEID    NPROC  SIZE   RSS MEMORY      TIME  CPU ZONE
     0       47  177M  104M    11%   0:00:31 0.1% global
     5       33  302M  244M    25%   0:01:12 0.0% gallery
     3       40  161M   91M   9.2%   0:00:40 0.0% nakos
     4       43  171M   94M   9.5%   0:00:44 0.0% mcdougallfamily
     2       30   96M   56M   5.6%   0:00:23 0.0% shared
     1       32  113M   60M   6.0%   0:00:45 0.0% packer
     7       43  203M   87M   8.7%   0:00:55 0.0% si
Total: 336 processes, 1202 lwps, load averages: 0.02, 0.01, 0.01
```

3.3 Process Status: **ps**

The standard command to list process information is `ps`, process status. Solaris ships with two versions: `/usr/bin/ps`, which originated from SVR4; and `/usr/ucb/ps`, originating from BSD. Sun has enhanced the SVR4 version since its inclusion with Solaris, in particular allowing users to select their own output fields.

3.3.1 `/usr/bin/ps` Command

The `/usr/bin/ps` command lists a line for each process.

```
$ ps -ef
     UID   PID  PPID   C    STIME TTY         TIME CMD
    root     0     0   0   Feb 08 ?           0:02 sched
    root     1     0   0   Feb 08 ?           0:15 /sbin/init
    root     2     0   0   Feb 08 ?           0:00 pageout
    root     3     0   1   Feb 08 ?         163:12 fsflush
  daemon   238     1   0   Feb 08 ?           0:00 /usr/lib/nfs/statd
    root     7     1   0   Feb 08 ?           4:58 /lib/svc/bin/svc.startd
    root     9     1   0   Feb 08 ?           1:35 /lib/svc/bin/svc.configd
    root   131     1   0   Feb 08 ?           0:39 /usr/sbin/pfild
  daemon   236     1   0   Feb 08 ?           0:11 /usr/lib/nfs/nfsmapid
...
```

`ps -ef` prints every process (`-e`) with full details (`-f`).
The following fields are printed by `ps -ef`:

- **`UID`.** The user name for the effective owner UID.
- **`PID`.** Unique process ID for this process.
- **`PPID`.** Parent process ID.
- **`C`.** The man page reads "Processor utilization for scheduling (obsolete)." This value now is recent percent CPU for a thread from the process and is read from `procfs` as `psinfo->pr_lwp->pr_cpu`. If the process is single threaded, this value represents recent percent CPU for the entire process (as with `pr_pctcpu`; see Section 2.12.3). If the process is multithreaded, then the value is from a recently running thread (selected by `prchoose()` from `uts/common/fs/proc/prsubr.c`); in that case, it may be more useful to run `ps` with the `-L` option, to list all threads.
- **`STIME`.** Start time for the process. This field can contain either one or two words, for example, `03:10:02` or `Feb 15`. This can annoy shell or Perl programmers who expect `ps` to produce a simple whitespace-delimited output. A fix is to use the `-o stime` option, which uses underscores instead of spaces, for example, `Feb_15`; or perhaps a better way is to write a C program and read the `procfs` structs directly.
- **`TTY`.** The controlling terminal for the process. This value is retrieved from `procfs` as `psinfo->pr_ttydev`. If the process was not created from a terminal, such as with daemons, `pr_ttydev` is set to `PRNODEV` and the `ps` command prints "`?`". If `pr_ttydev` is set to a device that `ps` does not understand, `ps` prints "`??`". This can happen when `pr_ttydev` is a `ptm` device (pseudo tty-master), such as with `dtterm` console windows.

- **TIME.** CPU-consumed time for the process. The units are in minutes and seconds of CPU runtime and originate from microstate accounting (user + system time). A large value here (more than several minutes) means either that the process has been running for a long time (check STIME) or that the process is hogging the CPU, possibly due to an application fault.
- **CMD.** The command that created the process and arguments, up to a width of 80 characters. It is read from procfs as psinfo->pr_psargs, and the width is defined in /usr/include/sys/procfs.h as PRARGSZ. The full command line does still exist in memory; this is just the truncated view that procfs provides.

For reference, Table 3.2 lists useful options for /usr/bin/ps.

Table 3.2 Useful /usr/bin/ps Options

Option	Description
-c	Print scheduling class and priority.
-e	List every process.
-f	Print full details; this is a standard selection of columns.
-l	Print long details, a different selection of columns.
-L	Print details by lightweight process (LWP).
-o format	Customize output fields.
-p proclist	Only examine these PIDs.
-u uidlist	Only examine processes owned by these user names or UIDs.
-Z	Print zone name.

Many of these options are straightforward. Perhaps the most interesting is -o, with which you can customize the output by selecting which fields to print. A quick list of the selectable fields is printed as part of the usage message.

```
$ ps -o
ps: option requires an argument -- o
usage: ps [ -aAdeflcjLPyZ ] [ -o format ] [ -t termlist ]
        [ -u userlist ] [ -U userlist ] [ -G grouplist ]
        [ -p proclist ] [ -g pgrplist ] [ -s sidlist ] [ -z zonelist ]
  'format' is one or more of:
        user ruser group rgroup uid ruid gid rgid pid ppid pgid sid taskid ctid
        pri opri pcpu pmem vsz rss osz nice class time etime stime zone zoneid
        f s c lwp nlwp psr tty addr wchan fname comm args projid project pset
```

The following example demonstrates the use of `-o` to produce an output similar to `/usr/ucb/ps aux`, along with an extra field for the number of threads (`NLWP`).

```
$ ps -eo user,pid,pcpu,pmem,vsz,rss,tty,s,stime,time,nlwp,comm
    USER   PID %CPU %MEM  VSZ  RSS TT      S    STIME        TIME NLWP COMMAND
    root     0  0.0  0.0    0    0 ?       T   Feb_08       00:02    1 sched
    root     1  0.0  0.1 2384  408 ?       S   Feb_08       00:15    1 /sbin/init
    root     2  0.0  0.0    0    0 ?       S   Feb_08       00:00    1 pageout
    root     3  0.4  0.0    0    0 ?       S   Feb_08    02:45:59    1 fsflush
  daemon   238  0.0  0.0 2672    8 ?       S   Feb_08       00:00    1 /usr/lib/nfs/statd
...
```

A brief description for each of the selectable fields is in the man page for `ps`. The following extra fields were selected in this example:

- **%CPU.** Percentage of recent CPU usage. This is based on `pr_pctcpu`, See Section 2.12.3.
- **%MEM.** Ratio of RSS over the total number of usable pages in the system (`total_pages`). Since RSS is an approximation that includes shared memory, this percentage is also an approximation and may overcount memory. It is possible for the `%MEM` column to sum to over 100%.
- **VSZ.** Total virtual memory size for the mappings within the process, including all mapped files and devices, in kilobytes.
- **RSS.** Approximation for the physical memory used by the process, in kilobytes. See Section 6.7.
- **S.** State of the process: on a processor (`O`), on a run queue (`R`), sleeping (`S`), zombie (`Z`), or being traced (`T`).
- **NLWP.** Number of lightweight processes associated with this process; since Solaris 9 this equals the number of user threads.

The `-o` option also allows the headers to be set (for example, `-o user=USERNAME`).

3.3.2 /usr/ucb/ps

This version of `ps` is often used with the following options.

```
$ /usr/ucb/ps aux
USER       PID %CPU %MEM   SZ  RSS TT       S    START  TIME COMMAND
root         3  0.5  0.0    0    0 ?        S   Feb 08 166:25 fsflush
root     15861  0.3  0.2 1352  920 pts/3    O 12:47:16  0:00 /usr/ucb/ps aux
root     15862  0.2  0.2 1432 1048 pts/3    S 12:47:16  0:00 more
root      5805  0.1  0.3 2992 1504 pts/3    S   Feb 16  0:03 bash
root         7  0.0  0.5 7984 2472 ?        S   Feb 08  5:03 /lib/svc/bin/svc.s
root       542  0.0  0.1 7328  176 ?        S   Feb 08  4:25 /usr/apache/bin/ht
root         1  0.0  0.1 2384  408 ?        S   Feb 08  0:15 /sbin/init
...
```

Here we listed all processes (a), printed user-focused output (u), and included processes with no controlling terminal (x). Many of the columns print the same details (and read the same procfs values) as discussed in Section 3.3.1. There are a few key differences in the way this ps behaves:

- The output is sorted on %CPU, with the highest %CPU process at the top.
- The COMMAND field is truncated so that the output fits in the terminal window. Using ps auxw prints a wider output, truncated to a maximum of 132 characters. Using ps auxww prints the full command-line arguments with no truncation (something that /usr/bin/ps cannot do). This is fetched, if permissions allow, from /proc/<pid>/as.
- If the values in the columns are large enough they can collide. For example:

```
$ /usr/ucb/ps aux
USER       PID %CPU %MEM   SZ  RSS TT       S    START  TIME COMMAND
user1     3132  5.2  4.33132422084 pts/4    S   Feb 16 132:26 Xvnc :1 -desktop X
user1     3153  1.2  2.93544414648 ?        R   Feb 16 21:45 gnome-terminal --s
user1    16865  1.0 10.87992055464 pts/18   S   Mar 02 42:46 /usr/sfw/bin/../li
user1     3145  0.9  1.422216 7240 ?        S   Feb 16 17:37 metacity --sm-save
user1     3143  0.5  0.3 7988 1568 ?        S   Feb 16 12:09 gnome-smproxy --sm
user1     3159  0.4  1.425064 6996 ?        S   Feb 16 11:01 /usr/lib/wnck-appl
...
```

This can make both reading and postprocessing the values quite difficult.

3.4 Tools for Listing and Controlling Processes

Solaris provides a set of tools for listing and controlling processes. The general syntax is as follows:

```
$ ptool pid
$ ptool pid/lwpid
```

The following is a summary for each. Refer to the man pages for additional details.

3.4.1 Process Tree: ptree

The process parent-child relationship can be displayed with the ptree command. By default, all processes within the same process group ID are displayed. See

Section 2.12 in *Solaris™ Internals* for information about how processes are grouped in Solaris.

```
$ ptree 22961
301   /usr/lib/ssh/sshd
  21571 /usr/lib/ssh/sshd
    21578 /usr/lib/ssh/sshd
      21580 -ksh
        22961 /opt/filebench/bin/filebench
          22962 shadow -a shadow -i 1 -s ffffffff10000000 -m /var/tmp/fbench9Ca
          22963 shadow -a shadow -i 2 -s ffffffff10000000 -m /var/tmp/fbench9Ca
          22964 shadow -a shadow -i 3 -s ffffffff10000000 -m /var/tmp/fbench9Ca
          22965 shadow -a shadow -i 4 -s ffffffff10000000 -m /var/tmp/fbench9Ca
...
```

3.4.2 Grepping for Processes: pgrep

The `pgrep` command provides a convenient way to produce a process ID list matching certain criteria.

```
$ pgrep filebench
22968
22961
22966
22979
...
```

The search term will do partial matching, which can be disabled with the `-x` option (exact match). The `-l` option lists matched process names.

3.4.3 Killing Processes: pkill

The `pkill` command provides a convenient way to send signals to a list or processes matching certain criteria.

```
$ pkill -HUP in.named
```

If the signal is not specified, the default is to send a `SIGTERM`.

Typing `pkill d` by accident as root may have a disastrous effect; it will match every process containing a “d” (which is usually quite a lot) and send them all a `SIGTERM`. Due to the way `pkill` doesn't use `getopt()` for the signal, aliasing isn’t perfect; and writing a shell function is nontrivial.

3.4.4 Temporarily Stop a Process: pstop

A process can be temporarily suspended with the `pstop` command.

```
$ pstop 22961
```

3.4.5 Making a Process Runnable: prun

A process can be made runnable with the `prun` command.

```
$ prun 22961
```

3.4.6 Wait for Process Completion: pwait

The `pwait` command blocks and waits for termination of a process.

```
$ pwait 22961
(sleep...)
```

3.4.7 Reap a Zombie Process: preap

A zombie process can be reaped with the `preap` command, which was added in Solaris 9.

```
$ preap 22961
(sleep...)
```

3.5 Process Introspection Commands

Solaris provides a set of utilities for inspecting the state of processes. Most of the introspection tools can be used either on a running process or postmortem on a core file resulting from a process dump. The general syntax is as follows:

```
$ ptool pid
$ ptool pid/lwpid
$ ptool core
```

See the man pages for each of these tools for additional details.

3.5.1 Process Stack: pstack

The stacks of all or specific threads within a process can be displayed with the `pstack` command.

```
$ pstack 23154
23154:  shadow -a shadow -i 193 -s ffffffff10000000 -m /var/tmp/fbench9Cai2S
-----------------  lwp# 1 / thread# 1  --------------------
 ffffffff7e7ce0f4 lwp_wait (2, ffffffff7fffe9cc)
 ffffffff7e7c9528 _thrp_join (2, 0, 0, 1, 100000000, ffffffff7fffe9cc) + 38
 0000000100018300 threadflow_init (ffffffff3722f1b0, ffffffff10000000, 10006a658, 0, 0,
1000888b0) + 184
 00000001000172f8 procflow_exec (6a000, 10006a000, 0, 6a000, 5, ffffffff3722f1b0) + 15c
 0000000100026558 main (a3400, ffffffff7ffff948, ffffffff7fffeff8, a4000, 0, 1) + 414
 000000010001585c _start (0, 0, 0, 0, 0, 0) + 17c
-----------------  lwp# 2 / thread# 2  --------------------
 000000010001ae90 flowoplib_hog (30d40, ffffffff651f3650, 30d40, ffffffff373aa3b8, 1,
2e906) + 68
 00000001000194a4 flowop_start (ffffffff373aa3b8, 0, 1, 0, 1, 1000888b0) + 408
 ffffffff7e7ccea0 _lwp_start (0, 0, 0, 0, 0, 0)
```

The `pstack` command can be very useful for diagnosing process hangs or the status of core dumps. By default it shows a stack backtrace for all the threads within a process. It can also be used as a crude performance analysis technique; by taking a few samples of the process stack, you can often determine where the process is spending most of its time.

You can also dump a specific thread's stacks by supplying the lwpid on the command line.

```
sol8$ pstack 26258/2
26258:  shadow -a shadow -i 62 -s ffffffff10000000 -m /var/tmp/fbenchI4aGkZ
-----------------  lwp# 2 / thread# 2  --------------------
 ffffffff7e7ce138 lwp_mutex_timedlock (ffffffff10000060, 0)
 ffffffff7e7c4e8c mutex_lock_internal (ffffffff10000060, 0, 0, 1000, ffffffff7e8eef80,
ffffffff7f402400) + 248
 000000010001da3c ipc_mutex_lock (ffffffff10000060, 1000888b0, 100088800, 88800,
100000000, 1) + 4
 0000000100019d94 flowop_find (ffffffff651e2278, 100088800, ffffffff651e2180, 88800,
100000000, 1) + 34
 000000010001b990 flowoplib_sempost (ffffffff3739a768, ffffffff651e2180, 0, 6ac00, 1,
1) + 4c
 00000001000194a4 flowop_start (ffffffff3739a768, 0, 1, 0, 1, 1000888b0) + 408
 ffffffff7e7ccea0 _lwp_start (0, 0, 0, 0, 0, 0)
```

3.5.2 Process Memory Map: pmap -x

The `pmap` command inspects a process, displaying every mapping within the process's address space. The amount of resident, nonshared anonymous, and locked memory is shown for each mapping. This allows you to estimate shared and private memory usage.

```
sol9$ pmap -x 102908
102908:   sh
Address   Kbytes Resident   Anon  Locked Mode   Mapped File
00010000      88       88      -       - r-x--  sh
00036000       8        8      8       - rwx--  sh
00038000      16       16     16       - rwx--    [ heap ]
FF260000      16       16      -       - r-x--  en_.so.2
FF272000      16       16      -       - rwx--  en_US.so.2
FF280000     664      624      -       - r-x--  libc.so.1
FF336000      32       32      8       - rwx--  libc.so.1
FF360000      16       16      -       - r-x--  libc_psr.so.1
FF380000      24       24      -       - r-x--  libgen.so.1
FF396000       8        8      -       - rwx--  libgen.so.1
FF3A0000       8        8      -       - r-x--  libdl.so.1
FF3B0000       8        8      8       - rwx--    [ anon ]
FF3C0000     152      152      -       - r-x--  ld.so.1
FF3F6000       8        8      8       - rwx--  ld.so.1
FFBFE000       8        8      8       - rw---    [ stack ]
--------   -----   -----   -----   ------
total Kb    1072    1032      56       -
```

This example shows the address space of a Bourne shell, with the executable at the top and the stack at the bottom. The total `Resident` memory is 1032 Kbytes, which is an approximation of physical memory usage. Much of this memory will be shared by other processes mapping the same files. The total `Anon` memory is 56 Kbytes, which is an indication of the private memory for this process instance.

You can find more information on interpreting `pmap -x` output in Section 6.8.

3.5.3 Process File Table: `pfiles`

A list of files open within a process can be obtained with the `pfiles` command.

```
sol10# pfiles 21571
21571:  /usr/lib/ssh/sshd
  Current rlimit: 256 file descriptors
   0: S_IFCHR mode:0666 dev:286,0 ino:6815752 uid:0 gid:3 rdev:13,2
      O_RDWR|O_LARGEFILE
      /devices/pseudo/mm@0:null
   1: S_IFCHR mode:0666 dev:286,0 ino:6815752 uid:0 gid:3 rdev:13,2
      O_RDWR|O_LARGEFILE
      /devices/pseudo/mm@0:null
   2: S_IFCHR mode:0666 dev:286,0 ino:6815752 uid:0 gid:3 rdev:13,2
      O_RDWR|O_LARGEFILE
      /devices/pseudo/mm@0:null
   3: S_IFCHR mode:0000 dev:286,0 ino:38639 uid:0 gid:0 rdev:215,2
      O_RDWR FD_CLOEXEC
      /devices/pseudo/crypto@0:crypto
   4: S_IFIFO mode:0000 dev:294,0 ino:13099 uid:0 gid:0 size:0
      O_RDWR|O_NONBLOCK FD_CLOEXEC
   5: S_IFDOOR mode:0444 dev:295,0 ino:62 uid:0 gid:0 size:0
      O_RDONLY|O_LARGEFILE FD_CLOEXEC  door to nscd[89]
      /var/run/name_service_door
   6: S_IFIFO mode:0000 dev:294,0 ino:13098 uid:0 gid:0 size:0
      O_RDWR|O_NONBLOCK FD_CLOEXEC
```

continues

```
  7: S_IFDOOR mode:0644 dev:295,0 ino:55 uid:0 gid:0 size:0
     O_RDONLY FD_CLOEXEC  door to keyserv[169]
     /var/run/rpc_door/rpc_100029.1
  8: S_IFCHR mode:0000 dev:286,0 ino:26793 uid:0 gid:0 rdev:41,134
     O_RDWR FD_CLOEXEC
     /devices/pseudo/udp@0:udp
  9: S_IFSOCK mode:0666 dev:292,0 ino:31268 uid:0 gid:0 size:0
     O_RDWR|O_NONBLOCK
       SOCK_STREAM
       SO_REUSEADDR,SO_KEEPALIVE,SO_SNDBUF(49152),SO_RCVBUF(49640)
       sockname: AF_INET6 ::ffff:129.146.238.66  port: 22
       peername: AF_INET6 ::ffff:129.146.206.91  port: 63374
 10: S_IFIFO mode:0000 dev:294,0 ino:13098 uid:0 gid:0 size:0
     O_RDWR|O_NONBLOCK
 11: S_IFIFO mode:0000 dev:294,0 ino:13099 uid:0 gid:0 size:0
     O_RDWR|O_NONBLOCK FD_CLOEXEC
```

The Solaris 10 version of `pfiles` prints path names if possible.

3.5.4 Execution Time Statistics for a Process: `ptime`

A process can be timed with the `ptime` command for accurate microstate accounting instrumentation.[1]

```
$ ptime sleep 1

real        1.203
user        0.022
sys         0.140
```

3.5.5 Process Signal Disposition: `psig`

A list of the signals and their current disposition can be displayed with `psig`.

```
sol8$ psig $$
15481:          -zsh
HUP             caught   0
INT             blocked,caught   0
QUIT            blocked,ignored
ILL             blocked,default
TRAP            blocked,default
ABRT            blocked,default
EMT             blocked,default
FPE             blocked,default
KILL            default
BUS             blocked,default
SEGV            blocked,default
SYS             blocked,default
```

continues

1. Most other time commands now source the same microstate-accounting-based times.

```
PIPE            blocked,default
ALRM            blocked,caught   0
TERM            blocked,ignored
USR1            blocked,default
USR2            blocked,default
CLD             caught   0
PWR             blocked,default
WINCH           blocked,caught   0
URG             blocked,default
POLL            blocked,default
STOP            default
```

3.5.6 Process Libraries: `pldd`

A list of the libraries currently mapped into a process can be displayed with `pldd`. This is useful for verifying which version or path of a library is being dynamically linked into a process.

```
sol8$ pldd $$
482764: -ksh
/usr/lib/libsocket.so.1
/usr/lib/libnsl.so.1
/usr/lib/libc.so.1
/usr/lib/libdl.so.1
/usr/lib/libmp.so.2
```

3.5.7 Process Flags: `pflags`

The `pflags` command shows a variety of status information for a process. Information includes the mode—32-bit or 64-bit—in which the process is running and the current state for each thread within the process (see Section 3.1 in *Solaris™ Internals* for information on thread state). In addition, the top-level function on each thread's stack is displayed.

```
sol8$ pflags $$
482764: -ksh
              data model = _ILP32  flags = PR_ORPHAN
  /1:         flags = PR_PCINVAL|PR_ASLEEP [ waitid(0x7,0x0,0xffbff938,0x7) ]
```

3.5.8 Process Credentials: `pcred`

The credentials for a process can be displayed with `pcred`.

```
sol8$ pcred $$
482764: e/r/suid=36413  e/r/sgid=10
        groups: 10 10512 570
```

3.5.9 Process Arguments: pargs

The full process arguments and optionally a list of the current environment settings can be displayed for a process with the `pargs` command.

```
$ pargs -ae 22961
22961:  /opt/filebench/bin/filebench
argv[0]: /opt/filebench/bin/filebench

envp[0]: _=/opt/filebench/bin/filebench
envp[1]: MANPATH=/usr/man:/usr/dt/man:/usr/local/man:/opt/SUNWspro/man:/ws/on998-
tools/teamware/man:/home/rmc/local/man
envp[2]: VISUAL=/bin/vi
...
```

3.5.10 Process Working Directory: pwdx

The current working directory of a process can be displayed with the `pwdx` command.

```
$ pwdx  22961
22961:  /tmp/filebench
```

3.6 Examining User-Level Locks in a Process

With the process lock statistics command, `plockstat(1M)`, you can observe hot lock behavior in user applications that use user-level locks. The `plockstat` command uses DTrace to instrument and measure lock statistics.

```
# plockstat -p 27088
^C
Mutex block

Count     nsec Lock                          Caller
-------------------------------------------------------------------------------
  102 39461866 libaio.so.1`__aio_mutex      libaio.so.1`_aio_lock+0x28
    4 21605652 libaio.so.1`__aio_mutex      libaio.so.1`_aio_lock+0x28
   11 19908101 libaio.so.1`__aio_mutex      libaio.so.1`_aio_lock+0x28
   12 16107603 libaio.so.1`__aio_mutex      libaio.so.1`_aio_lock+0x28
   10  9000198 libaio.so.1`__aio_mutex      libaio.so.1`_aio_lock+0x28
   14  5833887 libaio.so.1`__aio_mutex      libaio.so.1`_aio_lock+0x28
   10  5366750 libaio.so.1`__aio_mutex      libaio.so.1`_aio_lock+0x28
  120   964911 libaio.so.1`__aio_mutex      libaio.so.1`_aio_lock+0x28
   48   713877 libaio.so.1`__aio_mutex      libaio.so.1`_aio_lock+0x28
   52   575273 libaio.so.1`__aio_mutex      libaio.so.1`_aio_lock+0x28
   89   534127 libaio.so.1`__aio_mutex      libaio.so.1`_aio_lock+0x28
   14   427750 libaio.so.1`__aio_mutex      libaio.so.1`_aio_lock+0x28
    1   348476 libaio.so.1`__aio_mutex      libaio.so.1`_aio_req_add+0x228
```

continues

```
Mutex spin

Count     nsec Lock                         Caller
-------------------------------------------------------------------------------
    1 375967836 0x1000bab58                  libaio.so.1`_aio_req_add+0x110
  427   817144 libaio.so.1`__aio_mutex      libaio.so.1`_aio_lock+0x28
   18   272192 libaio.so.1`__aio_mutex      libaio.so.1`_aio_lock+0x28
  176   212839 libaio.so.1`__aio_mutex      libaio.so.1`_aio_lock+0x28
   36   203057 libaio.so.1`__aio_mutex      libaio.so.1`_aio_lock+0x28
   41   197392 libaio.so.1`__aio_mutex      libaio.so.1`_aio_lock+0x28
    3   100364 libaio.so.1`__aio_mutex      libaio.so.1`_aio_lock+0x28

Mutex unsuccessful spin

Count     nsec Lock                         Caller
-------------------------------------------------------------------------------
  222   323249 libaio.so.1`__aio_mutex      libaio.so.1`_aio_lock+0x28
   60   301223 libaio.so.1`__aio_mutex      libaio.so.1`_aio_lock+0x28
   24   295308 libaio.so.1`__aio_mutex      libaio.so.1`_aio_lock+0x28
   56   286114 libaio.so.1`__aio_mutex      libaio.so.1`_aio_lock+0x28
   99   282302 libaio.so.1`__aio_mutex      libaio.so.1`_aio_lock+0x28
   25   278939 libaio.so.1`__aio_mutex      libaio.so.1`_aio_lock+0x28
    1   241628 libaio.so.1`__aio_mutex      libaio.so.1`_aio_req_add+0x228
```

Solaris has two main types of user-level locks:

- **Mutex lock.** An exclusive lock. Only one person can hold the lock. A mutex lock attempts to spin (busy spin in a loop) while trying obtain the lock if the holder is running on a CPU, or blocks if the holder is not running or after trying to spin for a predetermined period.
- **Reader/Writer Lock.** A shared reader lock. Only one person can hold the write lock, but many people could hold a reader lock while there are no writers.

The statistics show the different types of locks and information about contention for each. In this example, we can see mutex-block, mutex-spin, and mutex-unsuccessful-spin. For each type of lock we can see the following:

- **Count.** The number of contention events for this lock
- **nsec.** The average amount of time for which the contention event occurred
- **Lock.** The address or symbol name of the lock object
- **Caller.** The library and function of the calling function

3.7 Tracing Processes

Several tools in Solaris can be used to trace the execution of a process, most notably `truss` and DTrace.

3.7.1 Using `truss` to Trace Processes

By default, truss traces system calls made on behalf of a process. It uses the /proc interface to start and stop the process, recording and reporting information on each traced event.

This intrusive behavior of truss may slow a target process down to less than half its usual speed. This may not be acceptable for the analysis of live production applications. Also, when the timing of a process changes, race-condition faults can either be relieved or created. Having the fault vanish during analysis is both annoying and ironic.[2] Worse is when the problem gains new complexities.[3]

truss was first written as a clever use of /proc, writing control messages to /proc/<pid>/ctl to manipulate execution flow for debugging. It has since been enhanced to trace LWPs and user-level functions. Over the years it has been an indispensable tool, and there has been no better way to get at this information.

DTrace now exists and can get similar information more safely. However truss will still be valuable for many situations. When you use truss for troubleshooting commands, speed is hardly an issue; of more interest are the system calls that failed and why. truss also provides many translations from flags into codes, allowing many system calls to be easily understood.

In the following example, we trace the system calls for a specified process ID. The trace includes the user LWP (thread) number, system call name, arguments and return codes for each system call.

```
$ truss -p 26274
/1:     lwp_wait(2, 0xFFFFFFFF7FFFEA4C) (sleeping...)
/2:     pread(11, "\0\0\002\0\0\001\0\0\0\0".., 504, 0) = 504
/2:     pread(11, "\0\0\002\0\0\001\0\0\0\0".., 504, 0) = 504
/2:     semget(16897864, 128, 0)                     = 8
/2:     semtimedop(8, 0xFFFFFFFF7DEFBDF4, 2, 0xFFFFFFFF7DEFBDE0) = 0
/2:     pread(11, "\0\0\002\0\0\001\0\0\0\0".., 504, 0) = 504
/2:     pread(11, "\0\0\002\0\0\001\0\0\0\0".., 504, 0) = 504
/2:     semget(16897864, 128, 0)                     = 8
/2:     semtimedop(8, 0xFFFFFFFF7DEFBDF4, 2, 0xFFFFFFFF7DEFBDE0) = 0
/2:     semget(16897864, 128, 0)                     = 8
/2:     semtimedop(8, 0xFFFFFFFF7DEFBDF4, 2, 0xFFFFFFFF7DEFBDE0) = 0
/2:     semget(16897864, 128, 0)                     = 8
/2:     semtimedop(8, 0xFFFFFFFF7DEFBDF4, 2, 0xFFFFFFFF7DEFBDE0) = 0
/2:     semget(16897864, 128, 0)                     = 8
/2:     semtimedop(8, 0xFFFFFFFF7DEFBDF4, 2, 0xFFFFFFFF7DEFBDE0) = 0
/2:     semget(16897864, 128, 0)                     = 8
/2:     semtimedop(8, 0xFFFFFFFF7DEFBDF4, 2, 0xFFFFFFFF7DEFBDE0) = 0
/2:     semget(16897864, 128, 0)                     = 8
/2:     semtimedop(8, 0xFFFFFFFF7DEFBDF4, 2, 0xFFFFFFFF7DEFBDE0) = 0
...
```

2. It may lead to the embarrassing situation in which truss is left running perpetually.
3. Don't truss Xsun; it can deadlock—we did warn you!

Optionally, we can use the `-c` flag to summarize rather than trace a process's system call activity.

```
$ truss -c -p 26274
^C
syscall               seconds   calls  errors
read                     .002      10
semget                   .012      55
semtimedop               .015      55
pread                    .017      45
                     --------  ------   ----
sys totals:              .047     165      0
usr time:               1.030
elapsed:                7.850
```

The `truss` command also traces functions that are visible to the dynamic linker (this excludes functions that have been locally scoped as a performance optimization—see the *Solaris Linker and Libraries Guide*).

In the following example, we trace the functions within the target binary by specifying the `-u` option (trace functions rather than system calls) and `a.out` (trace within the binary, exclude libraries).

```
$ truss -u a.out -p 26274
/2@2:       -> flowop_endop(0xffffffff3735ef80, 0xffffffff6519c0d0, 0x0, 0x0)
/2:     pread(11, "\0\0\002\0\0\001\0\0\0\0".., 504, 0) = 504
/2@2:       -> filebench_log(0x5, 0x10006ae30, 0x0, 0x0)
/2@2:     -> filebench_log(0x3, 0x10006a8a8, 0xffffffff3735ef80, 0xffffffff6519c0d0)
/2@2:     -> filebench_log(0x3, 0x10006a868, 0xffffffff3735ef80, 0xffffffff6519c380)
/2@2:     -> filebench_log(0x3, 0x10006a888, 0xffffffff3735ef80, 0xffffffff6519c380)
/2@2:     <- flowoplib_hog() = 0xffffffff3735ef80
/2@2:     -> flowoplib_sempost(0xffffffff3735ef80, 0xffffffff6519c380)
/2@2:       -> filebench_log(0x5, 0x10006afa8, 0xffffffff6519c380, 0x1)
/2@2:       -> flowop_beginop(0xffffffff3735ef80, 0xffffffff6519c380)
/2:     pread(11, "\0\0\002\0\0\001\0\0\0\0".., 504, 0) = 504
/2@2:       -> filebench_log(0x5, 0x10006aff0, 0xffffffff651f7c30, 0x1)
/2:     semget(16897864, 128, 0)                        = 8
/2:     semtimedop(8, 0xFFFFFFFF7DEFBDF4, 2, 0xFFFFFFFF7DEFBDE0) = 0
/2@2:       -> filebench_log(0x5, 0x10006b048, 0xffffffff651f7c30, 0x1)
/2@2:       -> flowop_endop(0xffffffff3735ef80, 0xffffffff6519c380, 0xffffffff651f7c30)
/2:     pread(11, "\0\0\002\0\0\001\0\0\0\0".., 504, 0) = 504
/2@2:     -> filebench_log(0x3, 0x10006a8a8, 0xffffffff3735ef80, 0xffffffff6519c380)
...
```

See `truss(1M)` for further information.

3.7.2 Using `apptrace` to Trace Processes

The `apptrace` command was added in Solaris 8 to trace calls to shared libraries while evaluating argument details. In some ways it is an enhanced version of an older command, sotruss. The Solaris 10 version of apptrace has been enhanced further, printing separate lines for the return of each function call.

In the following example, apptrace prints shared library calls from the date command.

```
$ apptrace date
-> date     -> libc.so.1:int atexit(int (*)() = 0xff3c0090)
<- date     -> libc.so.1:atexit()
-> date     -> libc.so.1:int atexit(int (*)() = 0x11558)
<- date     -> libc.so.1:atexit()
-> date     -> libc.so.1:char * setlocale(int = 0x6, const char * = 0x11568 "")
<- date     -> libc.so.1:setlocale() = 0xff05216e
-> date     -> libc.so.1:char * textdomain(const char * = 0x1156c "SUNW_OST_OSCMD")
<- date     -> libc.so.1:textdomain() = 0x23548
-> date     -> libc.so.1:int getopt(int = 0x1, char *const * = 0xffbffd04, const char *
= 0x1157c "a:u")
<- date     -> libc.so.1:getopt() = 0xffffffff
-> date     -> libc.so.1:time_t time(time_t * = 0x225c0)
<- date     -> libc.so.1:time() = 0x440d059e
...
```

To illustrate the capability of apptrace, examine the example output for the call to getopt(). The entry to getopt() can be seen after the library name it belongs to (libc.so.1); then the arguments to getopt() are printed. The option string is displayed as a string, "a:u".

apptrace can evaluate structs for function calls of interest. In this example, full details for calls to strftime() are printed.

```
$ apptrace -v strftime date
-> date    -> libc.so.1:size_t strftime(char * = 0x225c4 "", size_t = 0x400, const char
* = 0xff056c38 "%a %b %e %T %Z %Y", const struct tm * = 0xffbffc54)
        arg0 = (char *) 0x225c4 ""
        arg1 = (size_t) 0x400
        arg2 = (const char *) 0xff056c38 "%a %b %e %T %Z %Y"
        arg3 = (const struct tm *) 0xffbffc54 (struct tm) {
        tm_sec: (int) 0x1
        tm_min: (int) 0x9
        tm_hour: (int) 0xf
        tm_mday: (int) 0x7
        tm_mon: (int) 0x2
        tm_year: (int) 0x6a
        tm_wday: (int) 0x2
        tm_yday: (int) 0x41
        tm_isdst: (int) 0x1
        }
        return = (size_t) 0x1c
<- date     -> libc.so.1:strftime() = 0x1c
Tue Mar  7 15:09:01 EST 2006
$
```

This output provides insight into how an application is using library calls, perhaps identifying faults where invalid data was used.

3.7.3 Using DTrace to Trace Process Functions

DTrace can trace system activity by using many different providers, including `syscall` to track system calls, `sched` to trace scheduling events, and `io` to trace disk and network I/O events. We can gain a greater understanding of process behavior by examining how the system responds to process requests. The following sections illustrate this:

- Section 2.15
- Section 4.15
- Section 6.11

However DTrace can drill even deeper: user-level functions from processes can be traced down to the CPU instruction. Usually, however, just the function entry and return probes suffice.

By specifying the provider name as `pid`*n*, where *n* is the process ID, we can use DTrace to trace process functions. Here we trace function entry and return.

```
# dtrace  -F -p 26274 -n 'pid$target:::entry,pid$target:::return { trace(timestamp); }'

dtrace: description 'pid$target:::entry, pid$target:::return ' matched 8836 probes
CPU FUNCTION
 18  -> flowoplib_sempost                         862876225376388
 18    -> flowoplib_sempost                       862876225406704
 18      -> filebench_log                         862876225479188
 18        -> filebench_log                       862876225505012
 18        <- filebench_log                       862876225606436
 18      <- filebench_log                         862876225668788
 18      -> flowop_beginop                        862876225733408
 18        -> flowop_beginop                      862876225770304
 18          -> pread                             862876225860508
 18            -> _save_nv_regs                   862876225924036
 18            <- _save_nv_regs                   862876226011512
 18            -> _pread                          862876226056292
 18            <- _pread                          862876226780092
 18          <- pread                             862876226867256
 18          -> gethrtime                         862876226940056
 18          <- gethrtime                         862876227018644
 18        <- flowop_beginop                      862876227106272
 18      <- flowop_beginop                        862876227162292
...
```

Unlike `truss`, DTrace does not stop and start the process for each traced function; instead, DTrace collects data in per-CPU buffers which the `dtrace` command asynchronously reads. The overhead when using DTrace on a process does depend on the frequency of traced events but is usually less than that of `truss`.

3.7.4 Using DTrace to Aggregate Process Functions

When processes are traced as in the previous example, the output may rush by at an incredible pace. Using aggregations can condense information of interest. In the following example, the `dtrace` command aggregated the user-level function calls of `inetd` while a connection was established.

```
# dtrace -n 'pid$target:a.out::entry { @[probefunc] = count(); }' -p 252
dtrace: description 'pid$target:a.out::entry ' matched 159 probes
^C

...
  store_rep_vals                                                      2
  store_retrieve_rep_vals                                             2
  make_handle_bound                                                   6
  debug_msg                                                          42
  msg                                                                42
  isset_pollfd                                                       58
  find_pollfd                                                        71
```

In this example, `debug_msg()` was called 42 times. The column on the right counts the number of times a function was called while `dtrace` was running. If we drop the `a.out` in the probe description, `dtrace` traces function calls from all libraries as well as `inetd`.

3.7.5 Using DTrace to Peer Inside Processes

One of the powerful capabilities of DTrace is its ability to look inside the address space of a process and dereference pointers of interest. We demonstrate by continuing with the previous `inetd` example.

A function called `debug_msg()` sounds interesting if we were troubleshooting a problem. `inetd`'s `debug_msg()` takes a format string and variables as arguments and prints them to a log file if it exists (`/var/adm/inetd.log`). Since the log file doesn't exist on our server, `debug_msg()` tosses out the messages.

Without stopping or starting `inetd`, we can use DTrace to see what `debug_msg()` would have been writing. We have to know the prototype for `debug_msg()`, so we either read it from the source code or guess.

```
# dtrace -n 'pid$target:a.out:debug_msg:entry { trace(copyinstr(arg0)); }' -p 252
dtrace: description 'pid$target:a.out:debug_msg:entry ' matched 1 probe
CPU     ID                    FUNCTION:NAME
  0  52162                  debug_msg:entry   Exiting poll, returned: %d
  0  52162                  debug_msg:entry   Entering process_terminated_methods
  0  52162                  debug_msg:entry   Entering process_network_events
  0  52162                  debug_msg:entry   Entering process_nowait_req
  0  52162                  debug_msg:entry   Entering accept_connection
```

continues

```
CPU     ID                    FUNCTION:NAME
  0  52162                   debug_msg:entry   Entering run_method, instance: %s,
                                               method: %s
  0  52162                   debug_msg:entry   Entering read_method_context: inst: %s,
                                               method: %s, path: %s
  0  52162                   debug_msg:entry   Entering passes_basic_exec_checks
  0  52162                   debug_msg:entry   Entering contract_prefork
  0  52162                   debug_msg:entry   Entering contract_postfork
  0  52162                   debug_msg:entry   Entering get_latest_contract
...
```

The first argument (arg0) contains the format string, and copyinstr() pulls the string from userland to the kernel, where DTrace is tracing. Although the messages printed in this example are missing their variables, they illustrate much of what inetd is internally doing. It is not uncommon to find some form of debug functions left behind in applications, and DTrace can extract them in this way.

3.7.6 Using DTrace to Sample Stack Backtraces

When we discussed the pstack command (Section 3.5.1), we suggested a crude analysis technique, by which a few stack backtraces could be taken to see where the process was spending most of its time. DTrace can turn crude into precise by taking samples at a configurable rate, such as 1000 hertz.

The following example samples user stack backtraces at 1000 hertz, matching on the PID for inetd. This is quite a useful DTrace one-liner.

```
# dtrace -n 'profile-1000hz /pid == $target/ { @[ustack()] = count(); }' -p 252
dtrace: description 'profile-1000hz ' matched 1 probe
^C
...
              libc.so.1`_waitid+0x8
              libc.so.1`waitpid+0x68
              inetd`process_terminated_methods+0x74
              inetd`event_loop+0x19c
              inetd`start_method+0x190
              inetd`_start+0x108
               11

              libc.so.1`__pollsys+0x4
              libc.so.1`poll+0x7c
              inetd`event_loop+0x70
              inetd`start_method+0x190
              inetd`_start+0x108
               28

              libc.so.1`__fork1+0x4
              inetd`run_method+0x27c
              inetd`process_nowait_request+0x1c8
              inetd`process_network_events+0xac
              inetd`event_loop+0x220
              inetd`start_method+0x190
              inetd`_start+0x108
               53
```

The final stack backtrace was sampled the most, 53 times. By reading through the functions, we can determine where inetd was spending its on-CPU time.

Rather than sampling until Ctrl-C is pressed, DTrace allows us to specify an interval with ease. We added a tick-5sec probe in the following to stop sampling and exit after 5 seconds.

```
# dtrace -n 'profile-1000hz /pid == $target/ { @[ustack()] = count(); }
tick-5sec { exit(0); }' -p 252
```

3.8 Java Processes

The following sections should shed some light on what your Java applications are doing. Topics such as profiling and tracing are discussed.

3.8.1 Process Stack on a Java Virtual Machine: pstack

You can use the C++ stack unmangler with Java virtual machine (JVM) targets to show the stacks for Java applications. The c++filt utility is provided with the Sun Workshop compiler tools.

```
$ pstack 27494 |c++filt
27494:  /usr/bin/java -client -verbose:gc -Xbatch -Xss256k -XX:+AggressiveHeap
-----------------  lwp# 1 / thread# 1  --------------------
 ff3409b4 pollsys  (0, 0, ffbfe858, 0)
 ff2dcec8 poll     (0, 0, 1d4c0, 10624c00, 0, 0) + 7c
 fed316d4 int os_sleep(long long,int) (0, 1d4c0, 1, ff3, 372c0, 0) + 148
 fed2f6e4 int os::sleep(Thread*,long long,int) (372c0, 0, 1d4c0, 7, 4, ff14f934) + 284
 fedc21e0 JVM_Sleep (2, ff14dd24, 0, 1d4c0, ff1470dc, 372c0) + 260
 f8c0bc20 * java/lang/Thread.sleep(J)V+0
 f8c0bbc4 * java/lang/Thread.sleep(J)V+0
 f8c05764 * spec/jbb/JBButil.SecondsToSleep(J)V+11 (line 740)
 f8c05764 * spec/jbb/Company.displayResultTotals(ZZ)V+235 (line 651)
 f8c05764 * spec/jbb/JBBmain.DoARun(Lspec/jbb/Company;SSII)V+197 (line 277)
 f8c05764 * spec/jbb/JBBmain.DOIT(Lspec/jbb/infra/Factory/Container;)V+186 (line 732)
 f8c05764 * spec/jbb/JBBmain.main([Ljava/lang/String;)V+1220 (line 1019)
 f8c00218 * StubRoutines (1)
 fecd9f00 void JavaCalls::call_helper(JavaValue*,methodHandle*,JavaCallArgu-
ments*,Thread*) (1, 372c0, ffbff018, ffbfef50, ffbff01c, 0) + 5b8
 fedb8e84 jni_CallStaticVoidMethod (ff14dd24, ff1470dc, 3788c, 372c0, 0, 37488) + 514
 000123b4 main     (ff14a040, 576d1a, fed2a6d0, 2, 2, 1d8) + 1314
 00011088 _start   (0, 0, 0, 0, 0, 0) + 108
```

3.8.2 JVM Profiling

While the JVM has long included the `-Xrunhprof` profiling flag, the Java 2 Platform, Standard Edition (J2SE) 5.0 and later use the JVMTI for heap and CPU profiling. Usage information is obtained with the `java -Xrunhprof` command. This profiling flag includes a variety of options and returns a lot of data. As a result, using a large number of options can significantly impact application performance.

To observe locks, use the command in the following example. Note that setting `monitor=y` specifies that locks should be observed. Setting `msa=y` turns on Solaris microstate accounting (see Section 3.2.2, and Section 2.10.3 in *Solaris™ Internals*), and `depth=8` sets the depth of the stack displayed.

```
# java -Xrunhprof:cpu=times,monitor=y,msa=y,depth=8,file=path_to_result_file app_name
MONITOR DUMP BEGIN\
    THREAD 200000, trace 302389, status: CW\
    THREAD 200001, trace 300000, status: R\
    THREAD 201044, trace 302505, status: R\
.....
    MONITOR Ljava/lang/StringBuffer;\
        owner: thread 200058, entry count: 1\
        waiting to enter:\
        waiting to be notified:\
MONITOR DUMP END\
MONITOR TIME BEGIN (total = 2442 ms) Sat Nov  5 11:51:04 2005\
rank   self  accum   count trace monitor\
   1 64.51% 64.51%     364 302089 java.lang.Class (Java)\
   2 20.99% 85.50%     294 302094 java.lang.Class (Java)\
   3  9.94% 95.44%     128 302027 sun.misc.Launcher$AppClassLoader (Java)\
   4  4.17% 99.61%     164 302122 sun.misc.Launcher$AppClassLoader (Java)\
   5  0.30% 99.90%      46 302158 sun.misc.Launcher$AppClassLoader (Java)\
   6  0.05% 99.95%      14 302163 sun.misc.Launcher$AppClassLoader (Java)\
   7  0.03% 99.98%      10 302202 sun.misc.Launcher$AppClassLoader (Java)\
   8  0.02% 100.00%       4 302311 sun.misc.Launcher$AppClassLoader (Java)\
MONITOR TIME END\
```

This command returns verbose data, including all the call stacks in the Java process. Note two sections at the bottom of the output: the `MONITOR DUMP` and `MONITOR TIME` sections. The `MONITOR DUMP` section is a complete snapshot of all the monitors and threads in the system. `MONITOR TIME` is a profile of monitor contention obtained by measuring the time spent by a thread waiting to enter a monitor. Entries in this record are ranked by the percentage of total monitor contention time and a brief description of the monitor.

In previous versions of the JVM, one option is to dump all the stacks on the running VM by sending a `SIGQUIT` (signal number `3`) to the Java process with the `kill` command. This dumps the stacks for all VM threads to the standard error as shown below.

```
# kill -3 <pid>
Full thread dump Java HotSpot(TM) Client VM (1.4.1_06-b01 mixed mode):
"Signal Dispatcher" daemon prio=10 tid=0xba6a8 nid=0x7 waiting on condition
[0..0]
"Finalizer" daemon prio=8 tid=0xb48b8 nid=0x4 in Object.wait()
[f2b7f000..f2b7fc24]
        at java.lang.Object.wait(Native Method)
        - waiting on <f2c00490> (a java.lang.ref.ReferenceQueue$Lock)
        at java.lang.ref.ReferenceQueue.remove(ReferenceQueue.java:111)
        - locked <f2c00490> (a java.lang.ref.ReferenceQueue$Lock)
        at java.lang.ref.ReferenceQueue.remove(ReferenceQueue.java:127)
        at java.lang.ref.Finalizer$FinalizerThread.run(Finalizer.java:159)
"Reference Handler" daemon prio=10 tid=0xb2f88 nid=0x3 in Object.wait()
[facff000..facffc24]
        at java.lang.Object.wait(Native Method)
        - waiting on <f2c00380> (a java.lang.ref.Reference$Lock)
        at java.lang.Object.wait(Object.java:426)
        at java.lang.ref.Reference$ReferenceHandler.run(Reference.java:113)
        - locked <f2c00380> (a java.lang.ref.Reference$Lock)
"main" prio=5 tid=0x2c240 nid=0x1 runnable [ffbfe000..ffbfe5fc]
        at testMain.doit2(testMain.java:12)
        at testMain.main(testMain.java:64)
"VM Thread" prio=5 tid=0xb1b30 nid=0x2 runnable
"VM Periodic Task Thread" prio=10 tid=0xb9408 nid=0x5 runnable
"Suspend Checker Thread" prio=10 tid=0xb9d58 nid=0x6 runnable
```

If the top of the stack for a number of threads terminates in a monitor call, this is the place to drill down and determine what resource is being contended. Sometimes removing a lock that protects a hot structure can require many architectural changes that are not possible. The lock might even be in a third-party library over which you have no control. In such cases, multiple instances of the application are probably the best way to achieve scaling.

3.8.3 Tuning Java Garbage Collection

Tuning garbage collection (GC) is one of the most important performance tasks for Java applications. To achieve acceptable response times, you will often have to tune GC. Doing that requires you to know the following:

- Frequency of garbage collection events
- Whether Young Generation or Full GC is used
- Duration of the garbage collection
- Amount of garbage generated

To obtain this data, add the `-verbosegc`, `-XX:+PrintGCTimeStamps`, and `-XX:+PrintGCDetails` flags to the regular JVM command line.

```
1953.954: [GC [PSYoungGen: 1413632K->37248K(1776640K)] 2782033K->1440033K(3316736K),
0.3666410 secs]
2018.424: [GC [PSYoungGen: 1477376K->37584K(1760640K)] 2880161K->1473633K(3300736K),
0.3825016 secs]
2018.806: [Full GC [PSYoungGen: 37584K->0K(1760640K)] [ParOldGen: 1436049K-
>449978K(1540096K)] 147363
3K->449978K(3300736K) [PSPermGen: 4634K->4631K(16384K)], 5.3205801 secs]
2085.554: [GC [PSYoungGen: 1440128K->39968K(1808384K)] 1890106K->489946K(3348480K),
0.2442195 secs]
```

The preceding example indicates that at 2018 seconds a Young Generation GC cleaned 3.3 Gbytes and took .38 seconds to complete. This was quickly followed by a Full GC that took 5.3 seconds to complete.

On systems with many CPUs (or hardware threads), the increased throughput often generates significantly more garbage in the VM, and previous GC tuning may no longer be valid. Sometimes Full GC is generated where previously only Young Generation existed. Dump the GC details to a log file to confirm.

Avoid full GC whenever you can because it severely affects response time. Full GC is usually an indication that the Java heap is too small. Increase the heap size by using the `-Xmx` and `-Xms` options until Full GCs are no longer triggered. It is best to preallocate the heap by setting `-Xmx` and `-Xms` to the same value. For example, to set the Java heap to 3.5 Gbytes, add the `-Xmx3550m`, `-Xms3550m`, `-Xmn2g`, and `-Xss128k` options. The J2SE 1.5.0_06 release also introduced parallelism into the old GCs. Add the `-XX:+UseParallelOldGC` option to the standard JVM flags to enable this feature.

For Young Generation the number of parallel GC threads is the number of CPUs presented by the Solaris OS. On UltraSPARC T1 processor-based systems this equates to the number of threads. It may be necessary to scale back the number of threads involved in Young Generation GC to achieve response time constraints. To reduce the number of threads, you can set `XX:ParallelGCThreads=number_of_threads`.

A good starting point is to set the GC threads to the number of cores on the system. Putting it all together yields the following flags.

```
-Xmx3550m -Xms3550m -Xmn2g -Xss128k -XX:+UseParallelOldGC -XX:+UseParallelGC -XX:Paral-
lelGCThreads=8
-XX:+PrintGCDetails -XX:+PrintGCTimestamps
```

Older versions of the Java virtual machine, such as 1.3, do not have parallel GC. This can be an issue on CMT processors because GC can stall the entire VM. Parallel GC is available from 1.4.2 onward, so this is a good starting point for Java applications on multiprocessor-based systems.

3.8.4 Using DTrace on Java Applications

The J2SE 6 (code-named Mustang) release introduces DTrace support within the Java HotSpot virtual machine. The providers and probes included in the Mustang release make it possible for DTrace to collect performance data for applications written in the Java programming language.

The Mustang release contains two built-in DTrace providers: `hotspot` and `hotspot_jni`. All probes published by these providers are user-level statically defined tracing (USDT) probes, accessed by the PID of the Java HotSpot virtual machine process.

The `hotspot` provider contains probes related to the following Java HotSpot virtual machine subsystems:

- **VM life cycle probes.** For VM initialization and shutdown
- **Thread life cycle probes.** For thread start and stop events
- **Class-loading probes.** For class loading and unloading activity
- **Garbage collection probes.** For systemwide garbage and memory pool collection
- **Method compilation probes.** For indication of which methods are being compiled by which compiler
- **Monitor probes.** For all wait and notification events, plus contended monitor entry and exit events
- **Application probes.** For fine-grained examination of thread execution, method entry/method returns, and object allocation

All hotspot probes originate in the VM library (`libjvm.so`), and as such, are also provided from programs that embed the VM. The `hotspot_jni` provider contains probes related to the Java Native Interface (JNI), located at the entry and return points of all JNI methods. In addition, the DTrace `jstack()` action prints mixed-mode stack traces including both Java method and native function names.

As an example, the following D script (`usestack.d`) uses the DTrace `jstack()` action to print the stack trace.

```
#!/usr/sbin/dtrace -s

BEGIN { this->cnt = 0; }

syscall::pollsys:entry
/pid == $1 && tid == 1/
{
   this->cnt++;
   printf("\n\tTID: %d", tid);
```

continues

```
   jstack(50);
}

syscall:::entry
/this->cnt == 1/
{
   exit(0);
}
```

And the stack trace itself appears as follows.

```
# ./usejstack.d 1344 | c++filt
CPU     ID                    FUNCTION:NAME
 0    316                   pollsys:entry
   TID: 1
   libc.so.1`__pollsys+0xa
   libc.so.1`poll+0x52
   libjvm.so`int os_sleep(long long,int)+0xb4
   libjvm.so`int os::sleep(Thread*,long long,int)+0x1ce
   libjvm.so`JVM_Sleep+0x1bc
   java/lang/Thread.sleep
   dtest.method3
   dtest.method2
   dtest.method1
   dtest.main
   StubRoutines (1)
   libjvm.so`void JavaCalls::call_helper(JavaValue*,methodHandle*,JavaCallArgu-
ments*,Thread*)+0x1b5
   libjvm.so`void os::os_exception_wrapper(void(*)(JavaValue*,methodHandle*,JavaCallAr-
guments*,Thread*),JavaValue*,methodHandle*,Ja
vaCallArguments*,Thread*)+0x18
   libjvm.so`void JavaCalls::call(JavaValue*,methodHandle,JavaCallArgu-
ments*,Thread*)+0x2d
   libjvm.so`void jni_invoke_static(JNIEnv_*,JavaValue*,_jobject*,JNICallType,_
jmethodID*,JNI_ArgumentPush er*,Thread*)+0x214
   libjvm.so`jni_CallStaticVoidMethod+0x244
   java`main+0x642
   StubRoutines (1)
```

The command line shows that the output from this script was piped to the `c++filt` utility, which demangles C++ mangled names, making the output easier to read. The DTrace header output shows that the CPU number is 0, the probe number is 316, the thread ID (TID) is 1, and the probe name is `pollsys:entry`, where `pollsys` is the name of the system call. The stack trace frames appear from top to bottom in the following order: two system call frames, three VM frames, five Java method frames, and VMframes in the remainder.

For further information on using DTrace with Java applications, see Section 10.3.