4

Observability Tools

Operating systems have historically provided many tools for observing system software and hardware components. To the newcomer, the wide range of available tools suggested that everything—or at least everything important—could be observed. In reality, there were many gaps, and systems performance experts became skilled in the art of inference and interpretation: figuring out activity from indirect tools and statistics.

For example, network packets could be examined individually (sniffing), but disk I/O could not (at least, not remotely easily). Conversely, disk utilization (percent busy) was easily observable from operating system tools, but network interface utilization was not.

With the addition of tracing frameworks, especially dynamic tracing, everything can now be observed, and virtually any activity can be observed directly. This has had a profound effect on systems performance, making it possible to create hundreds of new observability tools (the potential number is unlimited).

This chapter describes the types of operating system observability tools, including key examples, and the frameworks upon which they are built. The focus here is the frameworks, including /proc, kstat, /sys, DTrace, and SystemTap. Many more tools that use these frameworks are introduced in later chapters, including Linux Performance Events (LPE) in Chapter 6, CPUs.

4.1 Tool Types

Performance observability tools can be categorized as providing *system-wide* or *per-process* observability, and most are based on either *counters* or *tracing*. These attributes are shown in Figure 4.1, along with tool examples.

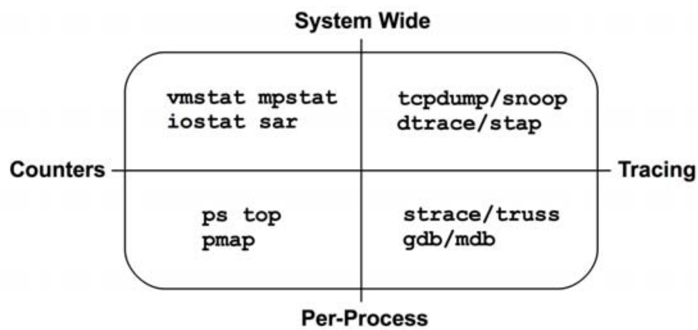

Figure 4-1 Observability tool types

Some tools fit in more than one quadrant; for example, `top(1)` also has a system-wide summary, and DTrace also has per-process capabilities.

There are also performance tools that are based on *profiling*. These observe activity by taking a series of snapshots either system-wide or per process.

The following sections summarize tools that use counters, tracing, and profiling as well as those that perform monitoring.

4.1.1 Counters

Kernels maintain various statistics, called *counters*, for counting events. They are usually implemented as unsigned integers that are incremented when events occur. For example, there are counters for the number of network packets received, disk I/O issued, and system calls performed.

Counters are considered "free" to use since they are enabled by default and maintained continually by the kernel. The only additional cost when using them is the act of reading their values from user-land (which should be negligible). The following example tools read these system-wide or per process.

System-Wide

These tools examine system-wide activity in the context of system software or hardware resources, using kernel counters. Examples are

- **vmstat:** virtual and physical memory statistics, system-wide
- **mpstat:** per-CPU usage
- **iostat:** per-disk I/O usage, reported from the block device interface
- **netstat:** network interface statistics, TCP/IP stack statistics, and some per-connection statistics
- **sar:** various statistics; can also archive them for historical reporting

These tools are typically viewable by all users on the system (non-root). Their statistics are also commonly graphed by monitoring software.

Many follow a usage convention where they accept an optional *interval* and *count*, for example, `vmstat(8)` with an interval of one second and an output count of three:

```
$ vmstat 1 3
procs -----------memory---------- ---swap-- -----io---- -system-- ----cpu----
 r  b   swpd   free   buff  cache   si   so    bi    bo   in    cs us sy id wa
 4  0      0 34455620 111396 13438564   0    0     0     5    1     2  0  0 100  0
 4  0      0 34458684 111396 13438588   0    0     0     0 2223 15198 13 11 76  0
 4  0      0 34456468 111396 13438588   0    0     0     0 1940 15142 15 11 74  0
```

The first line of output is the summary-since-boot, which shows averages for the entire time the system has been up. The subsequent lines are the one-second interval summaries, showing current activity. At least, this is the intent: this Linux version mixes summary-since-boot and current values for the first line.

Per-Process

These tools are process-oriented and use counters that the kernel maintains for each process. Examples are

- **ps:** process status, shows various process statistics, including memory and CPU usage.
- **top:** shows top processes, sorted by one of the statistics such as CPU usage. Solaris-based systems provide `prstat(1M)` for this purpose.
- **pmap:** lists process memory segments with usage statistics.

These tools typically read statistics from the /proc file system.

4.1.2 Tracing

Tracing collects per-event data for analysis. Tracing frameworks are not typically enabled by default, since tracing incurs CPU overhead to capture the data and can require significant storage to save it. These overheads can slow the target of tracing and need to be accounted for when interpreting measured times.

Logging, including the system log, can be thought of as low-frequency tracing that *is* enabled by default. Logging includes per-event data, although usually only for infrequent events such as errors and warnings.

The following are examples of system-wide and per-process tracing tools.

System-Wide

These tracing tools examine system-wide activity in the context of system software or hardware resources, using kernel tracing facilities. Examples are

- **`tcpdump`:** network packet tracing (uses libpcap)
- **`snoop`:** network packet tracing for Solaris-based systems
- **`blktrace`:** block I/O tracing (Linux)
- **`iosnoop`:** block I/O tracing (DTrace-based)
- **`execsnoop`:** tracing of new processes (DTrace-based)
- **`dtruss`:** system-wide buffered syscall tracing (DTrace-based)
- **DTrace:** tracing of kernel internals and the usage of any resource (not just network or block I/O), using static and dynamic tracing
- **SystemTap:** tracing of kernel internals and the usage of any resource, using static and dynamic tracing
- **perf:** Linux Performance Events, tracing static and dynamic probes

As DTrace and SystemTap are programming environments, system-wide tracing tools can be built upon them, including the few included in this list. More examples are provided throughout this book.

Per-Process

These tracing tools are process-oriented, as are the operating system frameworks on which they are based. Examples are

- **`strace`:** system call tracing for Linux-based systems
- **`truss`:** system call tracing for Solaris-based systems

- **`gdb`:** a source-level debugger, commonly used on Linux-based systems
- **`mdb`:** an extensible debugger for Solaris-based systems

The debuggers can examine per-event data, but they must do so by stopping and starting the execution of the target.

Tools such as DTrace, SystemTap, and perf all support a mode of execution where they can examine a single process only, although they are better described as system-wide tools.

4.1.3 Profiling

Profiling characterizes the target by collecting a set of samples or snapshots of its behavior. CPU usage is a common example, where samples are taken of the program counter or stack trace to characterize the code paths that are consuming CPU cycles. These samples are usually collected at a fixed rate, such as 100 or 1,000 Hz (cycles per second) across all CPUs. Profiling tools, or *profilers*, sometimes vary this rate slightly to avoid sampling in lockstep with target activity, which could lead to over- or undercounting.

Profiling can also be based on untimed hardware events, such as CPU hardware cache misses or bus activity. It can also show which code paths are responsible, information that can especially help developers optimize their code for the usage of system resources.

System-Wide and Per-Process

Here are some examples of profilers, all of which perform timer- and hardware-cache-based profiling:

- **`oprofile`:** Linux system profiling
- **`perf`:** a Linux performance toolkit, which includes profiling subcommands
- **DTrace:** programmatic profiling, timer-based using its `profile` provider, and hardware-event-based using its `cpc` provider
- **SystemTap:** programmatic profiling, timer-based using its `timer` tapset, and hardware-event-based using its `perf` tapset
- **`cachegrind`:** from the valgrind toolkit, can profile hardware cache usage and be visualized using kcachegrind
- **Intel VTune Amplifier XE:** Linux and Windows profiling, with a graphical interface including source browsing
- **Oracle Solaris Studio:** Solaris and Linux profiling with its Performance Analyzer, which has a graphical interface including source browsing

Programming languages often have their own special-purpose profilers that can inspect language context.

See Chapter 6, CPUs, for more about profiling tools.

4.1.4 Monitoring (sar)

Monitoring was introduced in Chapter 2, Methodology. The most commonly used tool for monitoring a single operating system host is the system activity reporter, `sar(1)`, originating from AT&T Unix. `sar(1)` is counter-based and has an agent that executes at scheduled times (via cron) to record the state of system counters. The `sar(1)` tool allows these to be viewed at the command line, for example:

```
# sar
Linux 3.2.6-3.fc16.x86_64 (web100)   04/15/2013      _x86_64_      (16 CPU)
05:00:00         CPU      %user     %nice   %system   %iowait    %steal      %idle
05:10:00         all      12.61      0.00      4.58      0.00      0.00      82.80
05:20:00         all      21.62      0.00      9.59      0.93      0.00      67.86
05:30:00         all      23.65      0.00      9.61      3.58      0.00      63.17
05:40:00         all      28.95      0.00      8.96      0.04      0.00      62.05
05:50:00         all      29.54      0.00      9.32      0.19      0.00      60.95
Average:         all      23.27      0.00      8.41      0.95      0.00      67.37
```

By default, `sar(1)` reads its statistics archive (if enabled) to print recent historic statistics. You can specify an optional interval and count for it to examine current activity at the rate specified.

Specific uses of `sar(1)` are described later in this book; see Chapters 6, 7, 8, 9, and 10. Appendix C is a summary of the `sar(1)` options.

While `sar(1)` can report many statistics, it may not cover all you really need, and those it does provide have at times been misleading (especially on Solaris-based systems [McDougall 06b]). Alternatives have been developed, such as System Data Recorder and Collectl.

In Linux, `sar(1)` is provided via the sysstat package. Third-party monitoring products are often built on `sar(1)`, or the same observability statistics it uses.

4.2 Observability Sources

The sections that follow describe various interfaces and frameworks that provide the statistics and data for observability tools. They are summarized in Table 4.1.

The main sources of systems performance statistics are covered next: /proc, /sys, and kstat. Delay accounting and microstate accounting are then described, and other sources are summarized. After these, the DTrace and SystemTap tools are introduced, which are built upon some of these frameworks.

Table 4.1 Observability Sources

Type	Linux	Solaris
Per-process counters	/proc	/proc, lxproc
System-wide counters	/proc, /sys	kstat
Device driver and debug info	/sys	kstat
Per-process tracing	ptrace, uprobes	procfs, dtrace
CPU performance counters	perf_event	libcpc
Network tracing	libpcap	libdlpi, libpcap
Per-thread latency metrics	delay accounting	microstate accounting
System-wide tracing	tracepoints, kprobes, ftrace	dtrace

4.2.1 /proc

This is a file system interface for kernel statistics. /proc contains a number of directories, where each directory is named after the process ID for the process it represents. These directories contain a number of files containing information and statistics about each process, mapped from kernel data structures. On Linux, there are additional files in /proc for system-wide statistics.

/proc is dynamically created by the kernel and is not backed by storage devices (it runs in-memory). It is mostly read-only, providing statistics for observability tools. Some files are writeable, for controlling process and kernel behavior.

The file system interface is convenient: it's an intuitive framework for exposing kernel statistics to user-land via the directory tree and has a well-known programming interface via the POSIX file system calls: `open()`, `read()`, `close()`. The file system also provides user-level security, through use of file access permissions.

The following shows how per-process statistics are read by `top(1)`, traced using `strace(1)`:

```
stat("/proc/14704", {st_mode=S_IFDIR|0555, st_size=0, ...}) = 0
open("/proc/14704/stat", O_RDONLY)      = 4
read(4, "14704 (sshd) S 1 14704 14704 0 -"..., 1023) = 232
close(4)
```

This has opened a file called `stat` in a directory named after the process ID, and then read the file contents.

`top(1)` repeats this for all active processes on the system. On some systems, especially those with many processes, the overhead from performing these can become noticeable, especially for versions of `top(1)` that repeat this sequence for every process on every screen update. This can lead to a situation where `top(1)` reports that `top(1)` itself is the highest CPU consumer!

The file system type for /proc on Linux is "proc" and for Solaris-based systems it is "procfs."

Linux

Various files are provided in /proc for per-process statistics. Here is an example of those that may be available:

```
$ ls -F /proc/28712
attr/            cpuset   io        mountinfo   oom_score    sessionid  syscall
auxv             cwd@     latency   mounts      pagemap      smaps      task/
cgroup           environ  limits    mountstats  personality  stack      wchan
clear_refs       exe@     loginuid  net/        root@        stat
cmdline          fd/      maps      numa_maps   sched        statm
coredump_filter  fdinfo/  mem       oom_adj     schedstat    status
```

The exact list of files available depends on the kernel version and CONFIG options.

Those related to per-process performance observability include

- **`limits`:** in-effect resource limits
- **`maps`:** mapped memory regions
- **`sched`:** various CPU scheduler statistics
- **`schedstat`:** CPU runtime, latency, and time slices
- **`smaps`:** mapped memory regions with usage statistics
- **`stat`:** process status and statistics, including total CPU and memory usage
- **`statm`:** memory usage summary in units of pages
- **`status`:** `stat` and `statm` information, human-readable
- **`task`:** directory of per-task statistics

Linux has also extended /proc to include system-wide statistics, contained in these additional files and directories:

```
$ cd /proc; ls -Fd [a-z]*
acpi/      dma          kallsyms    mdstat    schedstat   timer_list
buddyinfo  driver/      kcore       meminfo   scsi/       timer_stats
bus/       execdomains  keys        misc      self@       tty/
cgroups    fb           key-users   modules   slabinfo    uptime
cmdline    filesystems  kmsg        mounts@   softirqs    version
consoles   fs/          kpagecount  mtrr      stat        vmallocinfo
cpuinfo    interrupts   kpageflags  net@      swaps       vmstat
```

```
crypto     iomem        latency_stats  pagetypeinfo  sys/           zoneinfo
devices    ioports      loadavg        partitions    sysrq-trigger
diskstats  irq/         locks          sched_debug   sysvipc/
```

System-wide files related to performance observability include

- **cpuinfo:** physical processor information, including every virtual CPU, model name, clock speed, and cache sizes.
- **diskstats:** disk I/O statistics for all disk devices
- **interrupts:** interrupt counters per CPU
- **loadavg:** load averages
- **meminfo:** system memory usage breakdowns
- **net/dev:** network interface statistics
- **net/tcp:** active TCP socket information
- **schedstat:** system-wide CPU scheduler statistics
- **self:** a symlink to the current process ID directory, for convenience
- **slabinfo:** kernel slab allocator cache statistics
- **stat:** a summary of kernel and system resource statistics: CPUs, disks, paging, swap, processes
- **zoneinfo:** memory zone information

These are read by system-wide tools. For example, here's `vmstat(8)` reading /proc, as traced by `strace(1)`:

```
open("/proc/meminfo", O_RDONLY)         = 3
lseek(3, 0, SEEK_SET)                   = 0
read(3, "MemTotal:        889484 kB\nMemF"..., 2047) = 1170
open("/proc/stat", O_RDONLY)            = 4
read(4, "cpu  14901 0 18094 102149804 131"..., 65535) = 804
open("/proc/vmstat", O_RDONLY)          = 5
lseek(5, 0, SEEK_SET)                   = 0
read(5, "nr_free_pages 160568\nnr_inactive"..., 2047) = 1998
```

/proc files are usually text formatted, allowing them to be read easily from the command line and processed by shell scripting tools. For example:

```
$ cat /proc/meminfo
MemTotal:         889484 kB
MemFree:          636908 kB
Buffers:          125684 kB
Cached:            63944 kB
```

continues

```
SwapCached:            0 kB
Active:           119168 kB
[...]
$ grep Mem /proc/meminfo
MemTotal:         889484 kB
MemFree:          636908 kB
```

While this is convenient, it does add overhead for the kernel to encode the statistics as text, and for any user-land tool that then processes the text.

The contents of /proc are documented in the `proc(5)` man page and in the Linux kernel documentation: Documentation/filesystems/proc.txt. Some parts have extended documentation, such as `diskstats` in Documentation/iostats.txt and scheduler stats in Documentation/scheduler/sched-stats.txt. Apart from the documentation, you can also study the kernel source code to understand the exact origin of all items in /proc. It can also be helpful to read the source to the tools that consume them.

Some of the /proc entries depend on CONFIG options: `schedstats` are enabled with CONFIG_SCHEDSTATS, and `sched` with CONFIG_SCHED_DEBUG.

Solaris

On Solaris-based systems, /proc contains only process status statistics. System-wide observability is provided via other frameworks, mostly kstat.

Here is a list of files in a /proc process directory:

```
$ ls -F /proc/22449
as          cred  fd/      lstatus  map       path/   rmap    status  xmap
auxv        ctl   ldt      lusage   object/   priv    root@   usage
contracts/  cwd@  lpsinfo  lwp/     pagedata  psinfo  sigact  watch
```

Files related to performance observability include

- **`map:`** virtual address space mappings
- **`psinfo:`** miscellaneous process information, including CPU and memory usage
- **`status:`** process state information
- **`usage:`** extended process activity statistics, including process microstates, fault, block, context switch, and syscall counters
- **`lstatus:`** similar to `status`, but containing statistics for each thread
- **`lpsinfo:`** similar to `psinfo`, but containing statistics for each thread
- **`lusage:`** similar to `usage`, but containing statistics for each thread

- **lwpsinfo:** lightweight process (thread) statistics for the representative LWP (currently most active); there are also lwpstatus and lwpsinfo files
- **xmap:** extended memory mapping statistics (undocumented)

The following truss(1) output shows prstat(1M) reading status for a process:

```
open("/proc/4363/psinfo", O_RDONLY)            = 5
pread(5, "01\0\0\001\0\0\0\v11\0\0".., 416, 0)  = 416
```

The format of these files is binary, as seen by the pread() data above. psinfo contains

```
    typedef struct psinfo {
        int pr_flag;               /* process flags (DEPRECATED: see below) */
        int pr_nlwp;               /* number of active lwps in the process */
        int pr_nzomb;              /* number of zombie lwps in the process */
        pid_t pr_pid;              /* process id */
        pid_t pr_ppid;             /* process id of parent */
        pid_t pr_pgid;             /* process id of process group leader */
        pid_t pr_sid;              /* session id */
        uid_t pr_uid;              /* real user id */
        uid_t pr_euid;             /* effective user id */
        gid_t pr_gid;              /* real group id */
        gid_t pr_egid;             /* effective group id */
        uintptr_t pr_addr;         /* address of process */
        size_t pr_size;            /* size of process image in Kbytes */
        size_t pr_rssize;          /* resident set size in Kbytes */
        dev_t pr_ttydev;           /* controlling tty device (or PRNODEV) */
        ushort_t pr_pctcpu;        /* % of recent cpu time used by all lwps */
        ushort_t pr_pctmem;        /* % of system memory used by process */
        timestruc_t pr_start;      /* process start time, from the epoch */
        timestruc_t pr_time;       /* cpu time for this process */
        timestruc_t pr_ctime;      /* cpu time for reaped children */
        char pr_fname[PRFNSZ];     /* name of exec'ed file */
        char pr_psargs[PRARGSZ];   /* initial characters of arg list */
        int pr_wstat;              /* if zombie, the wait() status */
        int pr_argc;               /* initial argument count */
        uintptr_t pr_argv;         /* address of initial argument vector */
        uintptr_t pr_envp;         /* address of initial environment vector */
        char pr_dmodel;            /* data model of the process */
        lwpsinfo_t pr_lwp;         /* information for representative lwp */
        taskid_t pr_taskid;        /* task id */
        projid_t pr_projid;        /* project id */
        poolid_t pr_poolid;        /* pool id */
        zoneid_t pr_zoneid;        /* zone id */
        ctid_t pr_contract;        /* process contract id */
    } psinfo_t;
```

This can be read directly to a psinfo_t variable in user-space, where the members can then be dereferenced. This makes the Solaris /proc more suitable for processing by programs written in C, which can include the struct definitions from the system-supplied header files.

/proc is documented by the `proc(4)` man page, and by the sys/procfs.h header file. As with Linux, if the kernel is open source, it can be helpful to study the origin of these statistics and how tools consume them.

lxproc

There has been the occasional need for a Linux-like /proc on Solaris-based systems. One reason is for porting Linux observability tools (e.g., `htop(1)`), which can otherwise be difficult to port due to the /proc differences: from a text-based interface to binary.

One solution is the lxproc file system: it provides a loosely Linux-compatible /proc for Solaris-based systems and can be mounted in parallel with the standard procfs /proc. For example, lxproc can be mounted on /lxproc, and applications that require a Linux-like proc can be modified to load process information from /lxproc instead of /proc—what should be a minor change.

```
smartos# more /lxproc/meminfo
        total:     used:    free:  shared: buffers:  cached:
Mem:  1073741824 88395776 985346048        0        0        0
Swap: 2147483648 267640832 1879842816
MemTotal:   1048576 kB
MemFree:     962252 kB
[...]
```

Like Linux /proc, there are also directories for each process containing process information.

lxproc may be incomplete and require additions: it is provided only as a best-effort interface for simple Linux /proc users.

4.2.2 /sys

Linux provides a sysfs file system, mounted on /sys, which was introduced with the 2.6 kernel to provide a directory-based structure for kernel statistics. This differs from /proc, which has evolved over time and had various system statistics added to the top-level directory. sysfs was originally designed to provide device driver statistics but has been extended to include any statistic type.

For example, the following lists /sys files for CPU 0 (truncated):

```
$ find /sys/devices/system/cpu/cpu0 -type f
/sys/devices/system/cpu/cpu0/crash_notes
/sys/devices/system/cpu/cpu0/cache/index0/type
/sys/devices/system/cpu/cpu0/cache/index0/level
/sys/devices/system/cpu/cpu0/cache/index0/coherency_line_size
/sys/devices/system/cpu/cpu0/cache/index0/physical_line_partition
```

```
/sys/devices/system/cpu/cpu0/cache/index0/ways_of_associativity
/sys/devices/system/cpu/cpu0/cache/index0/number_of_sets
/sys/devices/system/cpu/cpu0/cache/index0/size
/sys/devices/system/cpu/cpu0/cache/index0/shared_cpu_map
/sys/devices/system/cpu/cpu0/cache/index0/shared_cpu_list
[...]
/sys/devices/system/cpu/cpu0/topology/physical_package_id
/sys/devices/system/cpu/cpu0/topology/core_id
/sys/devices/system/cpu/cpu0/topology/thread_siblings
/sys/devices/system/cpu/cpu0/topology/thread_siblings_list
/sys/devices/system/cpu/cpu0/topology/core_siblings
/sys/devices/system/cpu/cpu0/topology/core_siblings_list
```

Many of those listed provide information about the CPU hardware caches. The following output shows their contents (using `grep(1)`, so that the file name is included with the output):

```
$ grep . /sys/devices/system/cpu/cpu0/cache/index*/level
/sys/devices/system/cpu/cpu0/cache/index0/level:1
/sys/devices/system/cpu/cpu0/cache/index1/level:1
/sys/devices/system/cpu/cpu0/cache/index2/level:2
/sys/devices/system/cpu/cpu0/cache/index3/level:3
$ grep . /sys/devices/system/cpu/cpu0/cache/index*/size
/sys/devices/system/cpu/cpu0/cache/index0/size:32K
/sys/devices/system/cpu/cpu0/cache/index1/size:32K
/sys/devices/system/cpu/cpu0/cache/index2/size:256K
/sys/devices/system/cpu/cpu0/cache/index3/size:8192K
```

This shows that CPU 0 has access to two Level 1 caches, each 32 Kbytes, a Level 2 cache of 256 Kbytes, and a Level 3 cache of 8 Mbytes.

The /sys file system typically has tens of thousands of statistics in read-only files, as well as many writeable files for changing kernel state. For example, CPUs can be set to online or offline by writing "1" or "0" to a file named "online." As with reading statistics, setting state can be performed by using text strings at the command line (`echo 1 > filename`), rather than a binary interface.

4.2.3 kstat

Solaris-based systems have a kernel statistics (kstat) framework used by system-wide observability tools. kstat includes statistics for most resources, including CPUs, disks, network interfaces, memory, and many software components in the kernel. A typical system has tens of thousands of statistics available from kstat.

Unlike /proc or /sys, there is no pseudo file system for kstat, and it is read from /dev/kstat via `ioctl()`. This is usually performed via the libkstat library, which provides convenience functions, or via Sun::Solaris::Kstat, a Perl library for the same purpose (although it is being phased out in some distributions in favor of libkstat). The `kstat(1M)` tool provides the statistics at the command line and can be used with shell scripting.

kstats are structured as a four-tuple:

```
module:instance:name:statistic
```

These are

- ***module*:** This usually refers to the kernel module that created the statistic, such as `sd` for the SCSI disk driver, or `zfs` for the ZFS file system.
- ***instance*:** Some modules exist as multiple instances, such as an `sd` module for each SCSI disk. The instance is an enumeration.
- ***name*:** This is a name for the group of statistics.
- ***statistic*:** This is the individual statistic name.

For example, the following reads the `nproc` statistic using `kstat(1M)` and specifying the full four-tuple:

```
$ kstat -p unix:0:system_misc:nproc
unix:0:system_misc:nproc        94
```

This statistic shows the currently running number of processes. The `-p` option to `kstat(1M)` was used to print parseable output (colon-separated). A blank field is treated as a wildcard. Trailing colons can also be dropped. These rules together allow the following to match and print all statistics from the `system_misc` group:

```
$ kstat -p unix:0:system_misc
unix:0:system_misc:avenrun_15min        201
unix:0:system_misc:avenrun_1min 383
unix:0:system_misc:avenrun_5min 260
unix:0:system_misc:boot_time    1335893569
unix:0:system_misc:class        misc
unix:0:system_misc:clk_intr     1560476763
unix:0:system_misc:crtime       0
unix:0:system_misc:deficit      0
unix:0:system_misc:lbolt        1560476763
unix:0:system_misc:ncpus        2
unix:0:system_misc:nproc        94
unix:0:system_misc:snaptime     15604804.5606589
unix:0:system_misc:vac  0
```

The `avenrun*` statistics are used to calculate the system *load averages*, as reported by tools including `uptime(1)` and `top(1)`.

Many statistics in kstat are *cumulative*. Instead of providing the current value, they show the total since boot. For example:

```
$ kstat -p unix:0:vminfo:freemem
unix:0:vminfo:freemem   184882526123755
```

This freemem statistic is incremented per second with the number of free pages. This allows the average over time intervals to be calculated. The summary-since-boot, as printed by many system-wide observability tools, can also be calculated by dividing the current value by seconds since boot.

Another version of freemem provides the instantaneous value (`unix:0:system_pages:freemem`). This mitigates a shortcoming in the cumulative version: it takes at least one second to know the current value, so that would be the minimum time for which a delta could be calculated.

Without any statistic name, `kstat(1M)` lists all statistics. For example, the following commands pipe the list of all statistics into `grep(1)` to search for those containing `freemem`, and then `wc(1)` to count the number of total statistics:

```
$ kstat -p | grep freemem
unix:0:system_pages:freemem     5962178
unix:0:vminfo:freemem   184893612065859
$ kstat -p | wc -l
   33195
```

The kstat statistics are not formally documented because they are considered an *unstable interface*—subject to change whenever the kernel changes. To understand what each does, the locations that increment them can be studied in the kernel source (if available). For example, the cumulative freemem statistic originates from the following kernel code:

```
usr/src/uts/common/sys/sysinfo.h:
typedef struct vminfo {         /* (update freq) update action          */
        uint64_t freemem;       /* (1 sec) += freemem in pages          */
        uint64_t swap_resv;     /* (1 sec) += reserved swap in pages    */
        uint64_t swap_alloc;    /* (1 sec) += allocated swap in pages   */
        uint64_t swap_avail;    /* (1 sec) += unreserved swap in pages  */
        uint64_t swap_free;     /* (1 sec) += unallocated swap in pages */
        uint64_t updates;       /* (1 sec) ++                           */
} vminfo_t;

usr/src/uts/common/os/space.c:
vminfo_t        vminfo;         /* VM stats protected by sysinfolock mutex */

usr/src/uts/common/os/clock.c:
static void
```

continues

```
clock(void)
{
[...]
        if (one_sec) {
[...]
                vminfo.freemem += freemem;
```

The freemem statistic is incremented once per second in the kernel `clock()` routine, by the value of a global called `freemem`. Locations that modify `freemem` can be inspected to see all the code involved.

The source code to the existing system tools (if available) can also be studied for example kstat usage.

4.2.4 Delay Accounting

Linux systems with the CONFIG_TASK_DELAY_ACCT option track time per task in the following states:

- **Scheduler latency:** waiting for a turn on-CPU
- **Block I/O:** waiting for a block I/O to complete
- **Swapping:** waiting for paging (memory pressure)
- **Memory reclaim:** waiting for the memory reclaim routine

Technically, the scheduler latency statistic is sourced from schedstats (mentioned earlier, in /proc) but is exposed with the other delay accounting states. (It is in `struct sched_info`, not `struct task_delay_info`.)

These statistics can be read by user-level tools using taskstats, which is a netlink-based interface for fetching per-task and process statistics. The kernel source Documentation/accounting directory has both the documentation, delay-accounting.txt, and an example consumer, getdelays.c:

```
$ ./getdelays -dp 17451
print delayacct stats ON
PID     17451

CPU             count     real total  virtual total    delay total  delay average
                  386     3452475144    31387115236     1253300657          3.247ms
IO              count    delay total  delay average
                  302     1535758266              5ms
SWAP            count    delay total  delay average
                    0              0              0ms
RECLAIM         count    delay total  delay average
                    0              0              0ms
```

Times are in nanoseconds unless specified otherwise. This example was taken from a heavily CPU-loaded system, and the process inspected was suffering scheduler latency.

4.2.5 Microstate Accounting

Solaris-based systems have per-thread and per-CPU *microstate accounting*, which records a set of high-resolution times for predefined states. These were a vast improvement of accuracy over the prior tick-based metrics and also provided additional states for performance analysis [McDougall 06b]. They are exposed to user-level tools via kstat for per-CPU metrics and /proc for per-thread metrics.

The CPU microstates are shown as the `usr`, `sys`, and `idl` columns of `mpstat(1M)` (see Chapter 6, CPUs). You can find them in the kernel code as CMS_USER, CMS_SYSTEM, and CMS_IDLE.

The thread microstates are visible as the `USR`, `SYS`, . . . columns from `prstat –m` and are summarized in Section 6.6.7, prstat of Chapter 6, CPUs.

4.2.6 Other Observability Sources

Various other observability sources include

- **CPU performance counters:** These are programmable hardware registers that provide low-level performance information, including CPU cycle counts, instruction counts, stall cycles, and so on. On Linux they are accessed via the perf_events interface and the `perf_event_open()` syscall and are consumed by tools including `perf(1)`. On Solaris-based systems they are accessed via libcpc and consumed by tools including `cpustat(1M)`. For more about these counters and tools, see Chapter 6, CPUs.
- **Per-process tracing:** This traces user-level software events, such as syscalls and function calls. It is usually expensive to perform, slowing the target. On Linux there is the `ptrace()` syscall for controlling process tracing, which is used by `strace(1)` for tracing syscalls. Linux also has uprobes for user-level dynamic tracing. Solaris-based systems trace syscalls using procfs and the `truss(1)` tool and dynamic tracing via DTrace.
- **Kernel tracing:** On Linux, tracepoints provide static kernel probes (originally *kernel makers*), and kprobes provide dynamic probes. Both of these are used by tracing tools such as ftrace, `perf(1)`, DTrace, and SystemTap. On Solaris-based systems, static and dynamic probes are provided by the dtrace kernel module. Both DTrace and SystemTap, consumers of kernel tracing, will be covered in the following sections, which also explain the terms *static* and *dynamic* probes.

- **Network sniffing:** These interfaces provide a way to capture packets from network devices for detailed investigations into packet and protocol performance. On Linux, sniffing is provided via the libpcap library and /proc/net/dev and is consumed by the `tcpdump(8)` tool. On Solaris-based systems sniffing is provided via the libdlpi library and /dev/net and is consumed by the `snoop(1M)` tool. A port of libpcap and `tcpdump(8)` has also been developed for Solaris-based systems. There are overheads, both CPU and storage, for capturing and examining all packets. See Chapter 10, Network, for more about network sniffing.
- **Process accounting:** This dates back to mainframes and the need to bill departments and users for their computer usage, based on the execution and runtime of processes. It exists in some form for both Linux- and Solaris-based systems and can sometimes be helpful for performance analysis at the process level. For example, the `atop(1)` tool uses process accounting to catch and display information from short-lived processes that would otherwise be missed when taking snapshots of /proc [1].
- **System calls:** Some system or library calls may be available to provide some performance metrics. These include `getrusage()`, a function call for processes to get their own resource usage statistics, including user- and system-time, faults, messages, and context switches. Solaris-based systems also have `swapctl()`,a system function for swap device management and statistics (Linux has /proc/swap).

If you are interested in how each of these works, you will find that documentation is usually available, intended for the developer who is building tools upon these interfaces.

And More

Depending on your kernel version and enabled options, even more observability sources may be available. Some are mentioned in later chapters of this book.

Here are a few more:

- **Linux:** I/O accounting, blktrace, timer_stats, lockstat, debugfs
- **Solaris:** extended accounting, flow accounting, Solaris Auditing

One technique to find such sources is to read the kernel code you are interested in observing and see what statistics or tracepoints have been placed there.

In some cases there may be no kernel statistics for what you are after. Apart from dynamic tracing, covered next, you may find that debuggers can fetch kernel variables to shed some light on an investigation. These include `gdb(1)` and mdb(1) (Solaris only). A similar and even more desperate approach is used by tools that open /dev/mem or /dev/kmem to read kernel memory directly.

Multiple observability sources with different interfaces can be a burden to learn and can be inefficient when their capabilities overlap. As DTrace has been part of the Solaris kernel since 2003, there have been efforts to move some old tracing frameworks to DTrace, and to serve all new tracing needs from it. This consolidation has been working very well and has simplified tracing on Solaris-based systems. We can hope that this trend continues, and that the future for both kernels brings fewer, yet more powerful, observability frameworks.

4.3 DTrace

DTrace is an observability framework that includes a programming language and a tool. This section summarizes DTrace basics, including dynamic and static tracing, probes, providers, D, actions, variables, one-liners, and scripting. It is intended as a DTrace primer, providing you with enough background for understanding its use later in this book, where it is used to extend performance observability on both Solaris- and Linux-based systems.

DTrace can observe all user- and kernel-level code via instrumentation points called *probes*. When probes are hit, arbitrary actions may be performed in its D language. Actions can include counting events, recording timestamps, performing calculations, printing values, and summarizing data. These actions can be performed in *real time*, while tracing is still enabled.

As an example of using DTrace for dynamic tracing, the following instruments the kernel ZFS (file system) spa_sync() function, showing the completion time and duration in nanoseconds (illumos kernel):

```
# dtrace -n 'fbt:zfs:spa_sync:entry { self->start = timestamp; }
    fbt:zfs:spa_sync:return /self->start/ { printf("%Y: %d ns",
    walltimestamp, timestamp - self->start); self->start = 0; }'
dtrace: description 'fbt:zfs:spa_sync:entry ' matched 2 probes
CPU     ID                    FUNCTION:NAME
  7  65353                  spa_sync:return 2012 Oct 30 00:20:27: 63849335 ns
 12  65353                  spa_sync:return 2012 Oct 30 00:20:32: 39754457 ns
 18  65353                  spa_sync:return 2012 Oct 30 00:20:37: 261013562 ns
  8  65353                  spa_sync:return 2012 Oct 30 00:20:42: 29800786 ns
 17  65353                  spa_sync:return 2012 Oct 30 00:20:47: 250368664 ns
 20  65353                  spa_sync:return 2012 Oct 30 00:20:52: 37450783 ns
 11  65353                  spa_sync:return 2012 Oct 30 00:20:57: 56010162 ns
[...]
```

The spa_sync() function flushes written data to the ZFS storage devices, causing bursts of disk I/O. It is of particular interest for performance analysis, as I/O can sometimes queue behind the issued disk I/O. Using DTrace, information about the rate at which `spa_sync()` fires, and the duration, can be immediately seen and studied. Thousands of other kernel functions can be studied in a similar way, by either printing per-event details or summarizing them.

A key difference of DTrace from other tracing frameworks (e.g., syscall tracing) is that DTrace is designed to be production-safe, with minimized performance overhead. One way it does this is by use of per-CPU kernel buffers, which improve memory locality, reduce cache coherency overheads, and can remove the need for synchronization locks. These buffers are also used to pass data to user-land at a gentle rate (by default, once per second), minimizing context switches. DTrace also provides a set of actions that can summarize and filter data in-kernel, which also reduces data overheads.

DTrace supports both *static* and *dynamic* tracing, each providing complementary functionality. Static probes have a documented and stable interface, and dynamic probes allow virtually unlimited observability as needed.

4.3.1 Static and Dynamic Tracing

One way to understand static and dynamic tracing is to examine the source and CPU instructions involved. Consider the following code from the kernel block device interface (illumos), usr/src/uts/common/os/bio.c:

```
/*
 * Mark I/O complete on a buffer, release it if I/O is asynchronous,
 * and wake up anyone waiting for it.
 */
void
biodone(struct buf *bp)
{
        if (bp->b_flags & B_STARTED) {
                DTRACE_IO1(done, struct buf *, bp);
                bp->b_flags &= ~B_STARTED;
        }
[...]
```

The `DTRACE_IO1` macro is an example of a *static probe*, which is added to the code before compilation. There is no visible example of dynamic probes in the source code, since these are added after compilation while the software is running.

The compiled instructions for this function are (truncated)

```
> biodone::dis
biodone:                        pushq  %rbp
biodone+1:                      movq   %rsp,%rbp
biodone+4:                      subq   $0x20,%rsp
biodone+8:                      movq   %rbx,-0x18(%rbp)
```

```
biodone+0xc:                    movq   %rdi,-0x8(%rbp)
biodone+0x10:                   movq   %rdi,%rbx
biodone+0x13:                   movl   (%rdi),%eax
biodone+0x15:                   testl  $0x2000000,%eax
[...]
```

When using *dynamic tracing* to probe the entry to the `biodone()` function, the first instruction is changed:

```
> biodone::dis
biodone:                        int    $0x3
biodone+1:                      movq   %rsp,%rbp
biodone+4:                      subq   $0x20,%rsp
biodone+8:                      movq   %rbx,-0x18(%rbp)
biodone+0xc:                    movq   %rdi,-0x8(%rbp)
biodone+0x10:                   movq   %rdi,%rbx
biodone+0x13:                   movl   (%rdi),%eax
biodone+0x15:                   testl  $0x2000000,%eax
[...]
```

The `int` instruction calls a soft interrupt, which is programmed to perform the dynamic tracing action. When dynamic tracing is disabled, the instruction is returned to its original state. This is *live patching* of the kernel address space, and the technique used can vary between processor types.

Instructions are added only when dynamic tracing is enabled. When it is not enabled, there are no additional instructions for instrumentation, and therefore no probe effect. This is described as *zero overhead when not in use*. The overhead when it is in use from the additional instructions is proportional to the rate at which the probes fire: the rate of events that are traced, and the actions they perform.

DTrace can dynamically trace the entry and return of functions, and any instruction in user-space. Since this dynamically builds probes from CPU instructions, which can vary between software releases, it is considered an *unstable interface*. Any DTrace one-liners or scripts based on dynamic tracing may need updating for newer releases of the software that they trace.

4.3.2 Probes

DTrace probes are named with a four-tuple:

```
provider:module:function:name
```

The *provider* is the collection of related probes, similar to a software library. The *module* and *function* are dynamically generated and specify the code location of the probe. The *name* is the name of the probe itself.

When specifying these, wildcards ("*") may be used. Leaving a field blank ("::") is equivalent to a wildcard (":*:"). Blank left fields may also be dropped from the probe specification (e.g., ":::BEGIN" == "BEGIN").

For example:

```
io:::start
```

is the start probe from the io provider. The module and function fields are left blank, so these will match all locations of the start probe.

4.3.3 Providers

The DTrace providers available depend on your DTrace and operating system version. They may include

- **syscall:** system call trap table
- **vminfo:** virtual memory statistics
- **sysinfo:** system statistics
- **profile:** sampling at arbitrary rates
- **sched:** kernel scheduling events
- **proc:** process-level events: create, exec, exit
- **io:** block device interface tracing (disk I/O)
- **pid:** user-level dynamic tracing
- **tcp:** TCP protocol events: connections, send and receive
- **ip:** IP protocol events: send and receive
- **fbt:** kernel-level dynamic tracing

There are many additional providers for higher-level languages: Java, JavaScript, Node.js, Perl, Python, Ruby, Tcl, and others.

Many of the providers are implemented using static tracing, so that they have a stable interface. It's preferable to use these (over dynamic tracing) where possible, so that your scripts work for different versions of the target software. The trade-off is that visibility is limited in comparison, as only the essentials are promoted to the stable interface, to minimize maintenance and the documentation burden.

4.3.4 Arguments

Probes can provide data via a set of variables called *arguments*. The use of arguments depends on the provider.

For example, the syscall provider provides entry and return probes for each system call. These set the following argument variables:

- **Entry:** `arg0`, ..., `argN`: arguments to system call
- **Return:** `arg0` or `arg1`: return value; `errno` is also set

The fbt and pid providers set arguments similarly, allowing the data passed and returned to kernel- or user-level functions to be examined.

To find out what the arguments are for each provider, refer to its documentation (you can also try `dtrace(1)` with the `–lv` options, which prints a summary).

4.3.5 D Language

The D language is awk-like and can be used in one-liners or scripts (the same as awk). DTrace statements have the form

```
probe_description /predicate/ { action }
```

The *action* is a series of optional semicolon-delimited statements that are executed when the probe fires. The *predicate* is an optional filtering expression.

For example, the statement

```
proc:::exec-success /execname == "httpd"/ { trace(pid); }
```

traces the `exec-success` probe from the `proc` provider and performs the printing action `trace(pid)` if the process name is equal to `"httpd"`. The `exec-success` probe is commonly used to trace the creation of new processes and instruments a successful `exec()` system call. The current process name is retrieved using the built-in variable `execname`, and the current process ID via `pid`.

4.3.6 Built-in Variables

Built-in variables can be used in calculations and predicates and can be printed using actions such as `trace()` and `printf()`. Commonly used built-ins are listed in Table 4.2.

Table 4.2 Commonly Used Built-in Variables

Variable	Description
`execname`	on-CPU process name (string)
`uid`	on-CPU user ID
`pid`	on-CPU process ID
`timestamp`	current time, nanoseconds since boot
`vtimestamp`	time thread was on-CPU, nanoseconds
`arg0..N`	probe arguments (uint64_t)
`args[0]..[N]`	probe arguments (typed)
`curthread`	pointer to current thread kernel structure
`probefunc`	function component of probe description (string)
`probename`	name component of probe description (string)
`curpsinfo`	current process information

4.3.7 Actions

Commonly used actions include those listed in Table 4.3.

Table 4.3 Commonly Used Actions

Action	Description
`trace(`*arg*`)`	print `arg`
`printf(`*format*`, `*arg*`, ...)`	print formatted string
`stringof(addr)`	return a string from a kernel address
`copyinstr(addr)`	return a string from a user-space address (this requires the kernel to perform a *copy in* from user-space to kernel-space)
`stack(`*count*`)`	print kernel-level stack trace, truncated if a count is provided
`ustack(`*count*`)`	print user-level stack trace, truncated if a count is provided
`func(pc)`	return a kernel function name, from the kernel program counter (pc)
`ufunc(pc)`	return a user function name, from the user program counter (pc)
`exit(status)`	exit DTrace and return status
`trunc(@agg, `*count*`)`	truncate the aggregation, either fully (delete all keys) or to the number of keys specified (count)
`clear(@agg)`	delete values from an aggregation (keep keys)
`printa(format, @agg)`	print aggregation, formatted

The last three actions listed are for a special variable type called an *aggregation*.

4.3.8 Variable Types

Table 4.4 summarizes the types of variables, listed in order of usage preference (aggregations are chosen first, then low to high overhead).

Table 4.4 Variable Types and Their Overhead

Type	Prefix	Scope	Overhead	Multi-CPU Safe	Example Assignment
Aggregation	`@`	global	low	yes	`@x = count();`
Aggregation with keys	`@[]`	global	low	yes	`@x[pid] = count();`
Clause-local	`this->`	clause instance	very low	yes	`this->x = 1;`
Thread-local	`self->`	thread	medium	yes	`self->x = 1;`
Scalar	none	global	low–medium	no	`x = 1;`
Associative array	none	global	medium–high	no	`x[y] = 1;`

The *thread-local* variable has a per-thread scope. This allows data, such as timestamps, to be easily associated with a thread.

The *clause-local* variable is used for intermediate calculations and is valid only during action clauses for the same probe description.

Multiple CPUs writing to the same scalar at the same time can lead to a corrupt variable state, hence the "no." It's unlikely, but has happened, and has been noticed for string scalars (leading to a corrupted string).

An *aggregation* is a special variable type that can be tallied per CPU and combined later for passing to user-land. These have the lowest overhead and are used for summarizing data in different ways.

Actions that populate aggregations are listed in Table 4.5.

Table 4.5 Aggregating Actions

Aggregating Action	Description
`count()`	count occurrences
`sum(value)`	sum value

continues

Table 4.5 Aggregating Actions (*Continued*)

Aggregating Action	Description
`min(value)`	record minimum of value
`max(value)`	record maximum of value
`quantize(value)`	record value as a power-of-two histogram
`lquantize(value, min, max, step)`	record value as a linear histogram, with minimum, maximum, and step provided
`llquantize(value, factor, min_magnitude, max_magnitude, steps)`	record value as a hybrid log/linear histogram

As an example of an aggregation and a histogram action, `quantize()`, the following shows the returned sizes for the `read()` syscall:

```
# dtrace -n 'syscall::read:return { @["rval (bytes)"] = quantize(arg0); }'
dtrace: description 'syscall::read:return ' matched 1 probe
^C
  rval (bytes)
           value  ------------- Distribution ------------- count
              -1 |                                         0
               0 |@@@@@@@@@@@@@@                           447
               1 |@@@                                      100
               2 |                                         5
               4 |                                         0
               8 |                                         2
              16 |                                         2
              32 |@@                                       53
              64 |                                         1
             128 |                                         0
             256 |                                         0
             512 |                                         4
            1024 |@                                        19
            2048 |                                         10
            4096 |@                                        34
            8192 |@@@@                                     130
           16384 |@@@@@@                                   170
           32768 |@@@@                                     125
           65536 |@@@@                                     114
          131072 |                                         5
          262144 |                                         5
          524288 |                                         0
```

This one-liner gathers statistics while tracing and prints a summary when `dtrace` ends, in this case, when Ctrl-C was typed. The first line of output, `dtrace: description ...`, is printed by default by `dtrace`, providing an indication of when tracing has begun.

The `value` column is the minimum size for the quantized range, and the `count` column is the occurrences for that range. The middle shows an ASCII representation of the distribution. In this case, the most frequently returned size was zero bytes, which occurred 447 times. Many of the returned reads were between 8,192 and 131,071 bytes, with 170 in the 16,384 to 32,767 range. This bimodal distribution would not have been noticed in a tool that reported only an average.

4.3.9 One-Liners

DTrace allows you to write concise and powerful one-liners like those I demonstrated earlier. Following are some more examples.

Trace `open()` system calls, printing the process name and file path name:

```
dtrace -n 'syscall::open:entry { printf("%s %s", execname, copyinstr(arg0)); }'
```

Note that Oracle Solaris 11 significantly modified the system call trap table (which is probed by DTrace to create the syscall provider), such that tracing `open()` on that system becomes

```
dtrace -n 'syscall::openat:entry { printf("%s %s", execname, copyinstr(arg1)); }'
```

Summarize CPU cross calls by process name:

```
dtrace -n 'sysinfo:::xcalls { @[execname] = count(); }'
```

Sample kernel-level stacks at 99 Hz:

```
dtrace -n 'profile:::profile-99 { @[stack()] = count(); }'
```

Many more DTrace one-liners are used throughout this book and are listed in Appendix D.

4.3.10 Scripting

DTrace statements can be saved to a file for execution, allowing much longer DTrace programs to be written.

For example, the bitesize.d script shows requested disk I/O sizes by process name:

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

#pragma D option quiet

dtrace:::BEGIN
{
        printf("Tracing... Hit Ctrl-C to end.\n");
}

io:::start
{
        this->size = args[0]->b_bcount;
        @Size[pid, curpsinfo->pr_psargs] = quantize(this->size);
}

dtrace:::END
{
        printf("\n%8s  %s\n", "PID", "CMD");
        printa("%8d  %S\n%@d\n", @Size);
}
```

Since this file begins with an interpreter line (`#!`), it can be made executable and then run from the command line.

The `#pragma` line sets quiet mode, which suppresses the default DTrace output (which was seen in the earlier `spa_sync()` example and consists of the `CPU`, `ID`, and `FUNCTION:NAME` columns).

The actual enabling in this script by the `io:::start` probe is straightforward. The `dtrace:::BEGIN` probe fires at the start to print an informational message, and `dtrace:::END` fires at the end to format and print the summary.

Here is some example output:

```
# ./bitesize.d
Tracing... Hit Ctrl-C to end.
^C

     PID  CMD
    3424  tar cf /dev/null .\0

           value  ------------- Distribution ------------- count
             512 |                                         0
            1024 |@@@                                      39
            2048 |@@@@@@                                   71
            4096 |@@@@@@@@@                                111
            8192 |@@@@@@@@@@@@@@@@@@@@@                    259
           16384 |                                         6
           32768 |@                                        8
           65536 |                                         0
```

While tracing, most of the disk I/O was requested by the `tar` command, with sizes shown above.

bitesize.d is from a collection of DTrace scripts called the DTraceToolkit, which can be found online.

4.3.11 Overheads

As has been mentioned, DTrace minimizes instrumentation overhead by use of per-CPU kernel buffers and in-kernel aggregation summaries. By default, it also passes data from kernel-space to user-space at a gentle asynchronous rate of once per second. It has various other features that reduce overhead and improve safety, including a routine whereby it will abort tracing if it detects that the system may have become unresponsive.

The overhead cost of performing tracing is relative to the frequency of traces and the actions they perform. Tracing block device I/O is typically so infrequent (1,000 I/O per second or less) that the overheads are negligible. On the other hand, tracing network I/O, when packet rates can reach millions per second, can cause significant overhead.

The action also comes at a cost. For example, I frequently sample kernel stacks at a rate of 997 Hz across all CPUs (using `stack()`) without a noticeable overhead. Sampling user-level stacks is more involved (using `ustack()`), for which I typically reduce the rate to 97 Hz.

There are also overheads when saving data into variables, especially associative arrays. While the use of DTrace typically comes without noticeable overhead, you do need to be aware that it is possible, and to use some caution.

4.3.12 Documentation and Resources

The reference for DTrace, which documents all actions, built-ins, and standard providers, is the *Dynamic Tracing Guide*, originally by Sun Microsystems and made freely available online [2]. For background on dynamic tracing, the problems it solves, and the evolution of DTrace, see [Cantrill 04] and [Cantrill 06].

Appendix D lists handy DTrace one-liners. Apart from their utility, they may be a useful reference for learning DTrace, one line at a time.

For a reference of scripts and strategy, see the text *DTrace: Dynamic Tracing in Oracle Solaris, Mac OS X and FreeBSD* [Gregg 11]. The scripts from this book are available online [3].

The DTraceToolkit contains over 200 scripts and is currently hosted on my home page [4]. Many of the scripts are wrapped in shell or Perl, to provide command-line options and behavior like other Unix tools, for example, `execsnoop`:

```
# execsnoop -h
USAGE: execsnoop [-a|-A|-ehjsvZ] [-c command]
       execsnoop                # default output
                -a              # print all data
                -A              # dump all data, space delimited
                -e              # safe output, parseable
                -j              # print project ID
                -s              # print start time, us
                -v              # print start time, string
                -Z              # print zonename
                -c command      # command name to snoop
  eg,
        execsnoop -v            # human readable timestamps
        execsnoop -Z            # print zonename
        execsnoop -c ls         # snoop ls commands only
```

GUIs have also been built upon DTrace, including Oracle ZFS Appliance Analytics and Joyent Cloud Analytics.