

Linux is used primarily for process, memory, and device-driver support for hardware and has been expanded to include power management. The Android runtime environment includes a core set of libraries as well as the Dalvik virtual machine. Software designers for Android devices develop applications in the Java language. However, rather than using the standard Java API, Google has designed a separate Android API for Java development. The Java class files are first compiled to Java bytecode and then translated into an executable file that runs on the Dalvik virtual machine. The Dalvik virtual machine was designed for Android and is optimized for mobile devices with limited memory and CPU processing capabilities.

The set of libraries available for Android applications includes frameworks for developing web browsers (webkit), database support (SQLite), and multimedia. The libc library is similar to the standard C library but is much smaller and has been designed for the slower CPUs that characterize mobile devices.

2.8 Operating-System Debugging

We have mentioned debugging frequently in this chapter. Here, we take a closer look. Broadly, **debugging** is the activity of finding and fixing errors in a system, both in hardware and in software. Performance problems are considered bugs, so debugging can also include **performance tuning**, which seeks to improve performance by removing processing **bottlenecks**. In this section, we explore debugging process and kernel errors and performance problems. Hardware debugging is outside the scope of this text.

2.8.1 Failure Analysis

If a process fails, most operating systems write the error information to a **log file** to alert system operators or users that the problem occurred. The operating system can also take a **core dump**—a capture of the memory of the process—and store it in a file for later analysis. (Memory was referred to as the “core” in the early days of computing.) Running programs and core dumps can be probed by a debugger, which allows a programmer to explore the code and memory of a process.

Debugging user-level process code is a challenge. Operating-system kernel debugging is even more complex because of the size and complexity of the kernel, its control of the hardware, and the lack of user-level debugging tools. A failure in the kernel is called a **crash**. When a crash occurs, error information is saved to a log file, and the memory state is saved to a **crash dump**.

Operating-system debugging and process debugging frequently use different tools and techniques due to the very different nature of these two tasks. Consider that a kernel failure in the file-system code would make it risky for the kernel to try to save its state to a file on the file system before rebooting. A common technique is to save the kernel’s memory state to a section of disk set aside for this purpose that contains no file system. If the kernel detects an unrecoverable error, it writes the entire contents of memory, or at least the kernel-owned parts of the system memory, to the disk area. When the system reboots, a process runs to gather the data from that area and write it to a crash

Kernighan's Law

“Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.”

dump file within a file system for analysis. Obviously, such strategies would be unnecessary for debugging ordinary user-level processes.

2.8.2 Performance Tuning

We mentioned earlier that performance tuning seeks to improve performance by removing processing bottlenecks. To identify bottlenecks, we must be able to monitor system performance. Thus, the operating system must have some means of computing and displaying measures of system behavior. In a number of systems, the operating system does this by producing *trace listings* of system behavior. All interesting events are logged with their time and important parameters and are written to a file. Later, an analysis program can process the log file to determine system performance and to identify bottlenecks and inefficiencies. These same traces can be run as input for a simulation of a suggested improved system. Traces also can help people to find errors in operating-system behavior.

Another approach to performance tuning uses single-purpose, interactive tools that allow users and administrators to question the state of various system components to look for bottlenecks. One such tool employs the UNIX command `top` to display the resources used on the system, as well as a sorted list of the “top” resource-using processes. Other tools display the state of disk I/O, memory allocation, and network traffic.

The **Windows Task Manager** is a similar tool for Windows systems. The task manager includes information for current applications as well as processes, CPU and memory usage, and networking statistics. A screen shot of the task manager appears in Figure 2.19.

Making operating systems easier to understand, debug, and tune as they run is an active area of research and implementation. A new generation of kernel-enabled performance analysis tools has made significant improvements in how this goal can be achieved. Next, we discuss a leading example of such a tool: the Solaris 10 DTrace dynamic tracing facility.

2.8.3 DTrace

DTrace is a facility that dynamically adds probes to a running system, both in user processes and in the kernel. These probes can be queried via the D programming language to determine an astonishing amount about the kernel, the system state, and process activities. For example, Figure 2.20 follows an application as it executes a system call (`ioctl()`) and shows the functional calls within the kernel as they execute to perform the system call. Lines ending with “U” are executed in user mode, and lines ending in “K” in kernel mode.

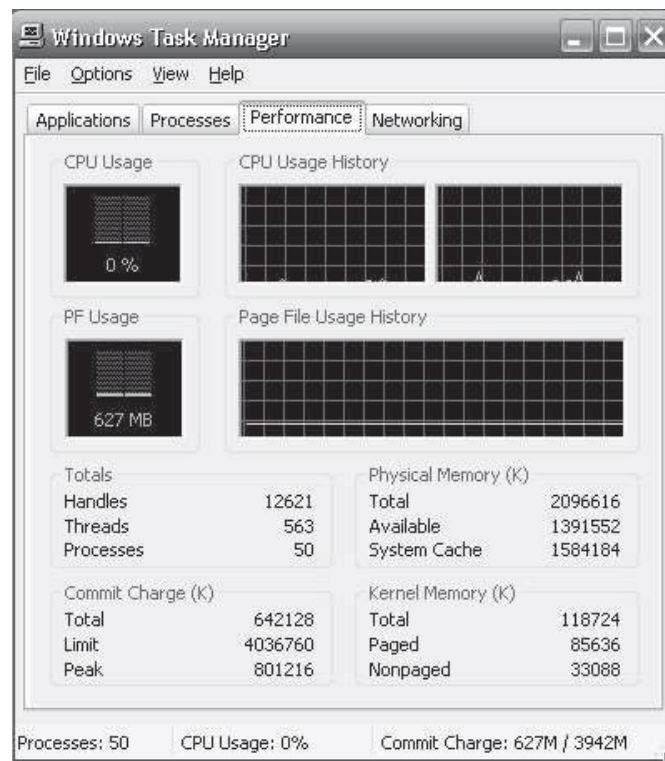


Figure 2.19 The Windows task manager.

Debugging the interactions between user-level and kernel code is nearly impossible without a toolset that understands both sets of code and can instrument the interactions. For that toolset to be truly useful, it must be able to debug any area of a system, including areas that were not written with debugging in mind, and do so without affecting system reliability. This tool must also have a minimum performance impact—ideally it should have no impact when not in use and a proportional impact during use. The DTrace tool meets these requirements and provides a dynamic, safe, low-impact debugging environment.

Until the DTrace framework and tools became available with Solaris 10, kernel debugging was usually shrouded in mystery and accomplished via happenstance and archaic code and tools. For example, CPUs have a breakpoint feature that will halt execution and allow a debugger to examine the state of the system. Then execution can continue until the next breakpoint or termination. This method cannot be used in a multiuser operating-system kernel without negatively affecting all of the users on the system. **Profiling**, which periodically samples the instruction pointer to determine which code is being executed, can show statistical trends but not individual activities. Code can be included in the kernel to emit specific data under specific circumstances, but that code slows down the kernel and tends not to be included in the part of the kernel where the specific problem being debugged is occurring.

```
# ./all.d 'pgrep xclock' XEventsQueued
dtrace: script './all.d' matched 52377 probes
CPU FUNCTION
0 -> XEventsQueued U
0 -> _XEventsQueued U
0 -> _X11TransBytesReadable U
0 <- _X11TransBytesReadable U
0 -> _X11TransSocketBytesReadable U
0 <- _X11TransSocketBytesreadable U
0 -> ioctl U
0 -> ioctl K
0 -> getf K
0 -> set_active_fd K
0 <- set_active_fd K
0 <- getf K
0 -> get_udatamodel K
0 <- get_udatamodel K
...
0 -> releasef K
0 -> clear_active_fd K
0 <- clear_active_fd K
0 -> cv_broadcast K
0 <- cv_broadcast K
0 <- releasef K
0 <- ioctl K
0 <- ioctl U
0 <- _XEventsQueued U
0 <- XEventsQueued U
```

Figure 2.20 Solaris 10 dtrace follows a system call within the kernel.

In contrast, DTrace runs on production systems—systems that are running important or critical applications—and causes no harm to the system. It slows activities while enabled, but after execution it resets the system to its pre-debugging state. It is also a broad and deep tool. It can broadly debug everything happening in the system (both at the user and kernel levels and between the user and kernel layers). It can also delve deep into code, showing individual CPU instructions or kernel subroutine activities.

DTrace is composed of a compiler, a framework, **providers** of **probes** written within that framework, and **consumers** of those probes. DTrace providers create probes. Kernel structures exist to keep track of all probes that the providers have created. The probes are stored in a hash-table data structure that is hashed by name and indexed according to unique probe identifiers. When a probe is enabled, a bit of code in the area to be probed is rewritten to call `dtrace_probe(probe identifier)` and then continue with the code's original operation. Different providers create different kinds of probes. For example, a kernel system-call probe works differently from a user-process probe, and that is different from an I/O probe.

DTrace features a compiler that generates a byte code that is run in the kernel. This code is assured to be “safe” by the compiler. For example, no loops are allowed, and only specific kernel state modifications are allowed when specifically requested. Only users with DTrace “privileges” (or “root” users)

are allowed to use DTrace, as it can retrieve private kernel data (and modify data if requested). The generated code runs in the kernel and enables probes. It also enables consumers in user mode and enables communications between the two.

A DTrace consumer is code that is interested in a probe and its results. A consumer requests that the provider create one or more probes. When a probe fires, it emits data that are managed by the kernel. Within the kernel, actions called **enabling control blocks**, or ECBs, are performed when probes fire. One probe can cause multiple ECBs to execute if more than one consumer is interested in that probe. Each ECB contains a predicate (“if statement”) that can filter out that ECB. Otherwise, the list of actions in the ECB is executed. The most common action is to capture some bit of data, such as a variable’s value at that point of the probe execution. By gathering such data, a complete picture of a user or kernel action can be built. Further, probes firing from both user space and the kernel can show how a user-level action caused kernel-level reactions. Such data are invaluable for performance monitoring and code optimization.

Once the probe consumer terminates, its ECBs are removed. If there are no ECBs consuming a probe, the probe is removed. That involves rewriting the code to remove the `dtrace_probe()` call and put back the original code. Thus, before a probe is created and after it is destroyed, the system is exactly the same, as if no probing occurred.

DTrace takes care to assure that probes do not use too much memory or CPU capacity, which could harm the running system. The buffers used to hold the probe results are monitored for exceeding default and maximum limits. CPU time for probe execution is monitored as well. If limits are exceeded, the consumer is terminated, along with the offending probes. Buffers are allocated per CPU to avoid contention and data loss.

An example of D code and its output shows some of its utility. The following program shows the DTrace code to enable scheduler probes and record the amount of CPU time of each process running with user ID 101 while those probes are enabled (that is, while the program runs):

```

sched::on-cpu
uid == 101
{
    self->ts = timestamp;
}

sched::off-cpu
self->ts
{
    @time[execname] = sum(timestamp - self->ts);
    self->ts = 0;
}

```

The output of the program, showing the processes and how much time (in nanoseconds) they spend running on the CPUs, is shown in Figure 2.21.

Because DTrace is part of the open-source OpenSolaris version of the Solaris 10 operating system, it has been added to other operating systems when those

```
# dtrace -s sched.d
dtrace: script 'sched.d' matched 6 probes
^C
      gnome-settings-d          142354
      gnome-vfs-daemon          158243
      dsdm                       189804
      wnck-applet                200030
      gnome-panel                277864
      clock-applet               374916
      mapping-daemon             385475
      xscreensaver               514177
      metacity                   539281
      Xorg                       2579646
      gnome-terminal             5007269
      mixer_applet2              7388447
      java                       10769137
```

Figure 2.21 Output of the D code.

systems do not have conflicting license agreements. For example, DTrace has been added to Mac OS X and FreeBSD and will likely spread further due to its unique capabilities. Other operating systems, especially the Linux derivatives, are adding kernel-tracing functionality as well. Still other operating systems are beginning to include performance and tracing tools fostered by research at various institutions, including the Paradyn project.

2.9 Operating-System Generation

It is possible to design, code, and implement an operating system specifically for one machine at one site. More commonly, however, operating systems are designed to run on any of a class of machines at a variety of sites with a variety of peripheral configurations. The system must then be configured or generated for each specific computer site, a process sometimes known as **system generation** **SYSGEN**.

The operating system is normally distributed on disk, on CD-ROM or DVD-ROM, or as an “ISO” image, which is a file in the format of a CD-ROM or DVD-ROM. To generate a system, we use a special program. This SYSGEN program reads from a given file, or asks the operator of the system for information concerning the specific configuration of the hardware system, or probes the hardware directly to determine what components are there. The following kinds of information must be determined.

- What CPU is to be used? What options (extended instruction sets, floating-point arithmetic, and so on) are installed? For multiple CPU systems, each CPU may be described.
- How will the boot disk be formatted? How many sections, or “partitions,” will it be separated into, and what will go into each partition?

- How much memory is available? Some systems will determine this value themselves by referencing memory location after memory location until an “illegal address” fault is generated. This procedure defines the final legal address and hence the amount of available memory.
- What devices are available? The system will need to know how to address each device (the device number), the device interrupt number, the device’s type and model, and any special device characteristics.
- What operating-system options are desired, or what parameter values are to be used? These options or values might include how many buffers of which sizes should be used, what type of CPU-scheduling algorithm is desired, what the maximum number of processes to be supported is, and so on.

Once this information is determined, it can be used in several ways. At one extreme, a system administrator can use it to modify a copy of the source code of the operating system. The operating system then is completely compiled. Data declarations, initializations, and constants, along with conditional compilation, produce an output-object version of the operating system that is tailored to the system described.

At a slightly less tailored level, the system description can lead to the creation of tables and the selection of modules from a precompiled library. These modules are linked together to form the generated operating system. Selection allows the library to contain the device drivers for all supported I/O devices, but only those needed are linked into the operating system. Because the system is not recompiled, system generation is faster, but the resulting system may be overly general.

At the other extreme, it is possible to construct a system that is completely table driven. All the code is always part of the system, and selection occurs at execution time, rather than at compile or link time. System generation involves simply creating the appropriate tables to describe the system.

The major differences among these approaches are the size and generality of the generated system and the ease of modifying it as the hardware configuration changes. Consider the cost of modifying the system to support a newly acquired graphics terminal or another disk drive. Balanced against that cost, of course, is the frequency (or infrequency) of such changes.

2.10 System Boot

After an operating system is generated, it must be made available for use by the hardware. But how does the hardware know where the kernel is or how to load that kernel? The procedure of starting a computer by loading the kernel is known as **booting** the system. On most computer systems, a small piece of code known as the **bootstrap program** or **bootstrap loader** locates the kernel, loads it into main memory, and starts its execution. Some computer systems, such as PCs, use a two-step process in which a simple bootstrap loader fetches a more complex boot program from disk, which in turn loads the kernel.

When a CPU receives a reset event—for instance, when it is powered up or rebooted—the instruction register is loaded with a predefined memory