MODERN OPERATING SYSTEMS

Fifth Edition

Andrew S. Tanenbaum
Herbert Bos

MODERN OPERATING SYSTEMS

FIFTH EDITION

ANDREW S. TANENBAUM
HERBERT BOS

Vrije Universiteit
Amsterdam, The Netherlands

10

CASE STUDY 1: UNIX, LINUX, AND ANDROID

In the previous chapters, we took a close look at many operating system principles, abstractions, algorithms, and techniques in general. Now it is time to look at some concrete systems to see how these principles are applied in the real world. We will begin with Linux, a popular variant of UNIX, which runs on a wide variety of computers. It is the dominant operating systems on high-end workstations and servers, but it is also used on systems ranging from smartphones (Android is based on Linux) to supercomputers.

Our discussion will start with its history and evolution of UNIX and Linux. Then we will provide an overview of Linux, to give an idea of how it is used. This overview will be of special value to readers familiar only with Windows, since the latter hides virtually all the details of the system from its users. Although graphical interfaces may be easy for beginners, they provide little flexibility and no insight into how the system works.

Next, we come to the heart of this chapter, an examination of processes, memory management, I/O, the file system, and security in Linux. For each topic, we will first discuss the fundamental concepts, then the system calls, and finally the implementation.

Right off the bat we should address the question: Why Linux? Linux is a variant of UNIX, but there are many other versions and variants of UNIX including AIX, FreeBSD, HP-UX, SCO UNIX, System V, Solaris, and others. Fortunately, the fundamental principles and system calls are pretty much the same for all of them (by design). Furthermore, the general implementation strategies, algorithms,

and data structures are similar, but there are some differences. To make the examples concrete, it is best to choose one of them and describe it consistently. Since most readers are more likely to have encountered Linux than any of the others, we will use it as our running example, but again be aware that except for the information on implementation, much of this chapter applies to all UNIX systems. A large number of books have been written on how to use UNIX, but there are also some about advanced features and system internals (Love, 2013; McKusick et al., 2014; Nemeth et al., 2013; Ostrowick, 2013; Sobell, 2014; Stevens and Rago, 2013; and Vahalia, 2007).

10.1 HISTORY OF UNIX AND LINUX

UNIX and Linux have a long and interesting history, so we will begin our study there. What started out as the pet project of one young researcher (Ken Thompson) has become a billion-dollar industry involving universities, multinational corporations, governments, and international standardization bodies. In the following pages, we will tell how this story has unfolded.

10.1.1 UNICS

Way back in the 1940s and 1950s, all computers were personal computers in the sense that the then-normal way to use a computer was to sign up for an hour of time and take over the entire machine for that period. Of course, these machines were physically immense, but only one person (the programmer) could use them at any given time. When batch systems took over, in the 1960s, the programmer submitted a job on punched cards by bringing it to the machine room. When enough jobs had been assembled, the operator read them all in as a single batch. It usually took an hour or more after submitting a job until the output was returned. Under these circumstances, debugging was a time-consuming process, because a single misplaced comma might result in wasting several hours of the programmer's time.

To get around what everyone viewed as an unsatisfactory, unproductive, and frustrating arrangement, timesharing was invented at Dartmouth College and M.I.T. The Dartmouth system ran only BASIC and enjoyed a short-term commercial success before vanishing. The M.I.T. system, CTSS, was general purpose and was a big success in the scientific community. Within a short time, researchers at M.I.T. joined forces with Bell Labs and General Electric (then a computer vendor) and began designing a second-generation system, **MULTICS** (**MULTiplexed Information and Computing Service**), as we discussed in Chap. 1.

Although Bell Labs was one of the founding partners in the MULTICS project, it later pulled out, which left one of the Bell Labs researchers, Ken Thompson, looking around for something interesting to do. He eventually decided to write a stripped-down MULTICS all by himself (in assembly language this time) on an old

discarded PDP-7 minicomputer. Despite the tiny size of the PDP-7, Thompson's system actually worked and could support Thompson's development effort. Consequently, one of the other researchers at Bell Labs, Brian Kernighan, somewhat jokingly called it **UNICS** (**UNiplexed Information and Computing Service**) because it supported only one user—Ken. Despite some puns about "EUNUCHS" being a castrated MULTICS, the name stuck, although the spelling was changed to **UNIX** later.

10.1.2 PDP-11 UNIX

Thompson's work so impressed his colleagues at Bell Labs that he was soon joined by Dennis Ritchie, and later by his entire department. Two major developments occurred around this time. First, UNIX was moved from the obsolete PDP-7 to the much more modern PDP-11/20 and then later to the PDP-11/45 and PDP-11/70. The latter two machines dominated the minicomputer world for much of the 1970s. The PDP-11/45 and PDP-11/70 were powerful machines with large physical memories for their era (256 KB and 2 MB, respectively). Also, they had memory-protection hardware, making it possible to support multiple users at the same time. However, they were both 16-bit machines that limited individual processes to 64 KB of instruction space and 64 KB of data space, even though the machine may have had far more physical memory.

The second development concerned the language in which UNIX was written. By now it was becoming painfully obvious that having to rewrite the entire system for each new machine was no fun at all, so Thompson decided to rewrite UNIX in a high-level language of his own design, called **B**. B was a simplified form of BCPL (which itself was a simplified form of CPL, which, like PL/I, never worked). Due to weaknesses in B, primarily lack of structures, this attempt was not successful. Ritchie then designed a successor to B, (naturally) called **C**, and wrote an excellent compiler for it. Working together, Thompson and Ritchie rewrote UNIX in C. C was the right language at the right time and has dominated system programming ever since.

In 1974, Ritchie and Thompson published a landmark paper about UNIX (Ritchie and Thompson, 1974). For the work described in this paper, they were later given the prestigious ACM Turing Award (Ritchie, 1984; Thompson, 1984). The publication of this paper stimulated many universities to ask Bell Labs for a copy of UNIX. Since Bell Labs' parent company, AT&T, was a regulated telephone monopoly at the time and was not permitted to be in the computer business, it had no objection to licensing UNIX to universities for a modest fee.

In one of those coincidences that often shape history, the PDP-11 was the computer of choice at nearly all university computer science departments, and the operating systems that came with the PDP-11 were widely regarded as dreadful by professors and students alike. UNIX quickly filled the void, not least because it was supplied with the complete source code, so that people could, and did, tinker with

it endlessly. Scientific meetings were organized around UNIX, with distinguished speakers getting up in front of the room to tell about some obscure kernel bug they had found and fixed. An Australian professor, John Lions, wrote a commentary on the UNIX source code of the type normally reserved for the works of Chaucer or Shakespeare (reprinted as Lions, 1996). The book described Version 6, so named because it was described in the sixth edition of the UNIX Programmer's Manual. The source code was 8200 lines of C and 900 lines of assembly code. As a result of all this activity, new ideas and improvements to the system spread rapidly.

Within a few years, Version 6 was replaced by Version 7, the first portable version of UNIX (it ran on the PDP-11 and the Interdata 8/32), by now 18,800 lines of C and 2100 lines of assembler. A whole generation of students was brought up on Version 7, which contributed to its spread after they graduated and went to work in industry. By the mid-1980s, UNIX was in widespread use on minicomputers and engineering workstations from a variety of vendors. A number of companies even licensed the source code to make their own version of UNIX. One of these was a small startup called Microsoft, which sold Version 7 under the name XENIX for a number of years until its interest turned elsewhere.

10.1.3 Portable UNIX

Now that UNIX was in C, moving it to a new machine, known as porting it, was much easier than in the early days when it was written in assembly language. A port requires first writing a C compiler for the new machine. Then it requires writing device drivers for the new machine's I/O devices, such as monitors, printers, and disks (which includes SSDs and other block storage devices). Although the driver code is in C, it cannot be moved to another machine, compiled, and run there because no two disks work the same way. Finally, a small amount of machine-dependent code, such as the interrupt handlers and memory-management routines, must be rewritten, usually in assembly language.

The first port beyond the PDP-11 was to the Interdata 8/32 minicomputer. This exercise revealed a large number of assumptions that UNIX implicitly made about the machine it was running on, such as the unspoken supposition that integers held 16 bits, pointers also held 16 bits (implying a maximum program size of 64 KB), and that the machine had exactly three registers available for holding important variables. None of these were true on the Interdata, so considerable work was needed to clean UNIX up.

Another problem was that although Ritchie's compiler was fast and produced good object code, it produced only PDP-11 object code. Rather than write a new compiler specifically for the Interdata, Steve Johnson of Bell Labs designed and implemented the **portable C compiler**, which could be retargeted to produce code for any reasonable machine with only a moderate amount of effort. For years, nearly all C compilers for machines other than the PDP-11 were based on Johnson's compiler, which greatly aided the spread of UNIX to new computers.

The port to the Interdata initially went slowly at first because the development work had to be done on the only working UNIX machine, a PDP-11, which was located on the fifth floor at Bell Labs. The Interdata was on the first floor. Generating a new version meant compiling it on the fifth floor and then physically carrying a magnetic tape down to the first floor to see if it worked. After several months of tape carrying, an unknown person said: "You know, we're the phone company. Can't we run a wire between these two machines?" Thus, was UNIX networking born. After the Interdata port, UNIX was ported to the VAX and later to other computers.

After AT&T was broken up in 1984 by the U.S. government, the company was legally free to set up a computer subsidiary, and did so. Shortly thereafter, AT&T released its first commercial UNIX product, System III. It was not well received, so it was replaced by an improved version, System V, a year later. Whatever happened to System IV is one of the great unsolved mysteries of computer science. The original System V has since been replaced by System V, releases 2, 3, and 4, each one bigger and more complicated than its predecessor. In the process, the original idea behind UNIX, of having a simple, elegant system, was gradually diminished. Although Ritchie and Thompson's group later produced an 8th, 9th, and 10th edition of UNIX, these were never widely circulated, as AT&T put all its marketing muscle behind System V. However, some of the ideas from the 8th, 9th, and 10th editions were eventually incorporated into System V. AT&T eventually decided that it wanted to be a telephone company after all, not a computer company, and sold its UNIX business to Novell in 1993. Novell subsequently sold it to the Santa Cruz Operation in 1995. By then it was almost irrelevant who owned it, since all the major computer companies already had licenses.

10.1.4 Berkeley UNIX

One of the many universities that acquired UNIX Version 6 early on was the University of California at Berkeley. Because the full source code was available, Berkeley was able to modify the system substantially. Aided by grants from ARPA, the U.S. Dept. of Defense's Advanced Research Projects Agency, Berkeley, produced and released an improved version for the PDP-11 called **1BSD** (**First Berkeley Software Distribution**). This tape was followed quickly by another, called **2BSD**, also for the PDP-11.

More important were **3BSD** and especially its successor, **4BSD** for the VAX. Although AT&T had a VAX version of UNIX, called **32V**, it was essentially Version 7. In contrast, 4BSD contained a large number of major improvements. Foremost among these was the use of virtual memory and paging, allowing programs to be larger than physical memory by paging parts of them in and out as needed. Another change allowed file names to be longer than 14 characters. The implementation of the file system was also changed, making it considerably faster. Signal handling was made more reliable. Networking was introduced, causing the

network protocol that was used, **TCP/IP**, to become a de facto standard in the UNIX world, and later in the Internet, which is dominated by UNIX-based servers.

Berkeley also added a substantial number of utility programs to UNIX, including a new editor (*vi*), a new shell (*csh*), Pascal and Lisp compilers, and many more. All these improvements caused Sun Microsystems, DEC, and other computer vendors to base their versions of UNIX on Berkeley UNIX, rather than on AT&T's "official" version, System V. As a consequence, Berkeley UNIX became well established in the academic, research, and defense worlds. For more information about Berkeley UNIX, see McKusick et al. (1996).

10.1.5 Standard UNIX

By the end of the 1980s, two different, and somewhat incompatible, versions of UNIX were in widespread use: 4.3BSD and System V Release 3. In addition, virtually every vendor added its own nonstandard enhancements. This split in the UNIX world, together with the fact that there were no standards for binary program formats, greatly inhibited the commercial success of UNIX because it was impossible for software vendors to write and package UNIX programs with the expectation that they would run on any UNIX system (as was routinely done with MS-DOS). Various attempts at standardizing UNIX initially failed. AT&T, for example, issued the **SVID** (**System V Interface Definition**), which defined all the system calls, file formats, and so on. This document was an attempt to keep all the System V vendors in line, but it had no effect on the enemy (BSD) camp, which just ignored it.

The first serious attempt to reconcile the two flavors of UNIX was initiated under the auspices of the IEEE Standards Board, a highly respected and, most importantly, neutral body. Hundreds of people from industry, academia, and government took part in this work. The collective name for this project was **POSIX**. The first three letters refer to Portable Operating System. The *IX* was added to make the name UNIXish.

After a great deal of argument and counter-argument, rebuttal and counter-rebuttal, the POSIX committee produced a standard known as **1003.1**. It defines a set of library procedures that every conformant UNIX system must supply. Most of these procedures invoke a system call, but a few can be implemented outside the kernel. Typical procedures are open, read, and fork. The idea of POSIX is that a software vendor who writes a program that uses only the procedures defined by 1003.1 knows that this program will run on every conformant UNIX system.

While it is true that most standards bodies tend to produce a horrible compromise with a few of everyone's pet features in it, 1003.1 is remarkably good considering the large number of parties involved and their respective vested interests. Rather than take the *union* of all features in System V and BSD as the starting point (the norm for most standards bodies), the IEEE committee took the *intersection*. Very roughly, if some feature was present in both System V and BSD, it was

included in the standard; otherwise it was not. As a consequence of this algorithm, 1003.1 bears a strong resemblance to the common ancestor of both System V and BSD, namely Version 7. The 1003.1 document is written in such a way that both operating system implementers and software writers can understand it, another novelty in the standards world, although work is already underway to remedy this.

Although the 1003.1 standard addresses only the system calls, related documents standardize threads, the utility programs, networking, and many other features of UNIX. In addition, the C language has also been standardized by ANSI and ISO.

10.1.6 MINIX

One property that all modern UNIX systems have is that they are large and complicated, in a sense the antithesis of the original idea behind UNIX. Even if the source code were freely available, which it is not in most cases, it is out of the question that a single person could understand it all anymore. This situation led one of the authors of this book (AST) to write a new UNIX-like system that was small enough to understand, was available with all the source code, and could be used for educational purposes. That system consisted of 11,800 lines of C and 800 lines of assembly code. Released in 1987, it was functionally almost equivalent to Version 7 UNIX, the mainstay of most computer science departments during the PDP-11 era.

MINIX was one of the first UNIX-like systems based on a microkernel design. The idea behind a microkernel is to provide minimal functionality in the kernel to make it reliable and efficient. Consequently, memory management and the file system were pushed out into user processes. The kernel handled message passing between the processes and little else. The kernel was 1600 lines of C and 800 lines of assembler. For technical reasons relating to the 8088 architecture, the I/O device drivers (2900 additional lines of C) were also in the kernel. The file system (5100 lines of C) and memory manager (2200 lines of C) ran as two separate user processes.

Microkernels have the advantage over monolithic systems that they are easy to understand and maintain due to their highly modular structure. Also, moving code from the kernel to user mode makes them highly reliable because the crash of a user-mode process does less damage than the crash of a kernel-mode component. Their main disadvantage is a slightly lower performance due to the extra switches between user mode and kernel mode. However, performance is not everything: all modern UNIX systems run X Windows in user mode and simply accept the performance hit to get the greater modularity (in contrast to Windows, where even the **GUI** (**Graphical User Interface**) is in the kernel). Other microkernels of this era were Mach (Accetta et al., 1986) and Chorus (Rozier et al., 1988).

Within a few months of its appearance, MINIX became a bit of a cult item, with its own USENET (now Google) newsgroup, *comp.os.minix*, and over 40,000

users. Numerous users contributed commands and other user programs, so MINIX quickly became a collective undertaking by large numbers of users over the Internet. It was a prototype of other collaborative efforts that came later. In 1997, Version 2.0 of MINIX was released and the base system, now including networking, had grown to 62,200 lines of code.

Around 2004, the direction of MINIX development changed sharply. The focus shifted to building an extremely reliable and dependable system that could automatically repair its own faults and become self-healing, continuing to function correctly even in the face of repeated software bugs being triggered. Consequently, the modularization idea present in Version 1 was greatly expanded in MINIX 3.0. Nearly all the device drivers were moved to user space, with each driver running as a separate process. The size of the entire kernel abruptly dropped to under 4000 lines of code, something a single programmer could easily understand. Internal mechanisms were changed to enhance fault tolerance in numerous ways.

In addition, over 650 popular UNIX programs were ported to MINIX 3.0, including the **X Window System** (sometimes just called **X**), various compilers (including *gcc*), text-processing software, networking software, Web browsers, and much more. Unlike previous versions, which were primarily educational in nature, starting with MINIX 3.0, the system was quite usable, with the focus moving toward high dependability. The ultimate goal is: No more reset buttons.

A third edition of the book *Operating Systems: Design and Implementation* appeared, describing the new system, giving its source code in an appendix, and describing it in detail (Tanenbaum and Woodhull, 2006). The system continues to evolve and has an active user community. It has since been ported to the ARM processor, making it available for embedded systems. For more details and to get the current version for free, you can visit *www.minix3.org*.

10.1.7 Linux

During the early years of MINIX development and discussion on the Internet, many people requested (or in many cases, demanded) more and better features, to which the author often said "No" (to keep the system small enough for students to understand completely in a one-semester university course). This continuous "No" irked many users. At this time, FreeBSD was not available, so that was not an option. After a number of years went by like this, a Finnish student, Linus Torvalds, decided to write another UNIX clone, named **Linux**, which would be a full-blown production system with many features MINIX was initially lacking. The first version of Linux, 0.01, was released in 1991. It was cross-developed on a MINIX machine and borrowed numerous ideas from MINIX, ranging from the structure of the source tree to the layout of the file system. However, it was a monolithic rather than a microkernel design, with the entire operating system in the kernel. The code totaled 9300 lines of C and 950 lines of assembler, which is

roughly similar to MINIX version in size and also comparable in functionality. De facto, it was a rewrite of MINIX, the only system Torvalds had source code for.

Linux rapidly grew in size and evolved into a full, production UNIX clone, as virtual memory, a more sophisticated file system, and many other features were added. Although it originally ran only on the 386 (and even had embedded 386 assembly code in the middle of C procedures), it was quickly ported to other platforms and now runs on a wide variety of machines, just as UNIX does. One difference with UNIX does stand out, however: Linux makes use of so many special features of the *gcc* compiler and would need a lot of work before it would compile with an ANSI standard C compiler. The shortsighted idea that *gcc* is the only compiler the world will ever see is already becoming a problem because the open-source LLVM compiler from the University of Illinois is rapidly gaining many adherents due to its flexibility and code quality. Since LLVM did not support all the nonstandard *gcc* extensions to C, it could not compile the Linux kernel without a lot of patches to the kernel to replace non-ANSI code when it was released. LLVM eventually supported some of the gcc extensions.

The next major release of Linux was version 1.0, issued in 1994. It was about 165,000 lines of code and included a new file system, memory-mapped files, and BSD-compatible networking with sockets and TCP/IP. It also included many new device drivers. Several minor revisions followed in the next two years.

By this time, Linux was sufficiently compatible with UNIX that a vast amount of UNIX software was ported to Linux, making it far more useful than it would have otherwise been. In addition, a large number of people were attracted to Linux and began working on the code and extending it in many ways under Torvalds' general supervision.

The next major release, 2.0, was made in 1996. It consisted of about 470,000 lines of C and 8000 lines of assembly code. It included support for 64-bit architectures, symmetric multiprogramming, new networking protocols, and numerous other features. A large fraction of the total code mass was taken up by an extensive collection of device drivers for an ever-growing set of supported peripherals. Additional releases followed frequently.

The version numbers of the Linux kernel consist of four numbers, *A.B.C.D*, such as 2.6.9.11. The first number denotes the kernel version. The second number denotes the major revision. Prior to the 2.6 kernel, even revision numbers corresponded to stable kernel releases, whereas odd ones corresponded to unstable revisions, under development. With the 2.6 kernel, that is no longer the case. The third number corresponds to minor revisions, such as support for new drivers. The fourth number corresponds to minor bug fixes or security patches. In July 2011, Linus Torvalds announced the release of Linux 3.0, not in response to major technical advances, but rather in honor of the 20th anniversary of the kernel. By early 2021, Linux 5.11 kernel version was released with over 30 million lines of code.

A large amount of standard UNIX software has been ported to Linux, including the popular X Window System and a great deal of networking software. Two

different GUIs (GNOME and KDE), which compete with each other, have also been written for Linux. In short, it has grown to a full-blown UNIX clone with all the bells and whistles a UNIX lover might conceivably want.

One unusual feature of Linux is its business model: it is free software. It can be downloaded from various sites on the Internet, for example: *www.kernel.org*. Linux comes with a license devised by Richard Stallman, founder of the Free Software Foundation. Despite the fact that Linux is free, this license, the **GPL** (**GNU Public License**), is longer than Microsoft's Windows license and specifies what you can and cannot do with the code. Users may use, copy, modify, and redistribute the source and binary code freely. The main restriction is that all works derived from the Linux kernel may not be sold or redistributed in binary form only; the source code must either be shipped with the product or be made available on request.

Although Torvalds still rides herd on the kernel fairly closely, a large amount of user-level software has been written by numerous other programmers, many of them having migrated over from the MINIX, BSD, and GNU online communities. However, as Linux evolves, an increasingly smaller fraction of the Linux community wants to hack source code (witness the hundreds of books telling how to install and use Linux and only a handful discussing the code or how it works). Also, many Linux users now forgo the free distribution on the Internet to buy one of the distributions available from numerous competing commercial companies. A popular Website listing the current top-100 Linux distributions is at *www.distrowatch.org*. As more and more software companies start selling their own versions of Linux and more and more hardware companies offer to preinstall it on the computers they ship, the line between commercial software and free software is beginning to blur substantially.

As a footnote to the Linux story, it is interesting to note that just as the Linux bandwagon was gaining steam, it got a big boost from a very unexpected source—AT&T. In 1992, Berkeley, by now running out of funding, decided to terminate BSD development with one final release, 4.4BSD (which later formed the basis of FreeBSD and also MacOS). Since this version contained essentially no AT&T code, Berkeley issued the software under an open source license (not GPL) that let everybody do whatever they wanted with it except one thing—sue the University of California. The AT&T subsidiary controlling UNIX promptly reacted by—you guessed it—suing the University of California. It also sued a company, BSDI, set up by the BSD developers to package the system and sell support, much as Red Hat and other companies now do for Linux. Since virtually no AT&T code was involved, the lawsuit was based on copyright and trademark infringement, including items such as BSDI's 1-800-ITS-UNIX telephone number. Although the case was eventually settled out of court, it kept FreeBSD off the market long enough for Linux to get well established. Had the lawsuit not happened, starting around 1993 there would have been serious competition between two free, open source UNIX systems: the reigning champion, BSD, a mature and stable system with a large

academic following dating back to 1977, versus the vigorous young challenger, Linux, just two years old but with a growing following among individual users. Who knows how this battle of the free UNICES would have turned out?

10.2 OVERVIEW OF LINUX

In this section, we will provide a general introduction to Linux and how it is used, for the benefit of readers not already familiar with it. Nearly all of this material applies to just about all UNIX variants with only small deviations. Although Linux has several graphical interfaces, the focus here is on how Linux appears to a programmer working in a shell window on X. Subsequent sections will focus on system calls and how it works inside.

10.2.1 Linux Goals

UNIX was always an interactive system designed to handle multiple processes and multiple users at the same time. It was designed by programmers, for programmers, to use in an environment in which the majority of the users are relatively sophisticated and are engaged in (often quite complex) software development projects. In many cases, a large number of programmers are actively cooperating to produce a single system, so UNIX has extensive facilities to allow people to work together and share information in controlled ways. The model of a group of experienced programmers working together closely to produce advanced software is obviously very different from the personal-computer model of a single beginner working alone with a word processor, and this difference is reflected throughout UNIX from start to finish. It is only natural that Linux inherited many of these goals, even though the first version was for a personal computer.

What is it that good programmers really want in a system? To start with, most like their systems to be simple, elegant, and consistent. For example, at the lowest level, a file should just be a collection of bytes. Having different classes of files for sequential access, random access, keyed access, remote access, and so on (as mainframes do) just gets in the way. Similarly, if the command

```
ls A*
```

means list all the files beginning with "A" then the command

```
rm A*
```

should mean remove all the files beginning with "A" and not remove the one file whose two-character name consists of an "A" and an asterisk. This characteristic is sometimes called the *principle of least surprise*.

Another thing that experienced programmers generally want is power and flexibility. This means that a system should have a small number of basic elements that can be combined in an infinite variety of ways to suit the application. One of the

basic guidelines behind Linux is that every program should do just one thing and do it well. Thus compilers do not produce listings, because other programs can do that better.

Finally, most programmers have a strong dislike for useless redundancy. Why type *copy* when *cp* is clearly enough to make it abundantly clear what you want? It is a complete waste of valuable hacking time. To extract all the lines containing the string "ard" from the file *f*, the Linux programmer merely types

```
grep ard f
```

The opposite approach is to have the programmer first select the *grep* program (with no arguments), and then have *grep* announce itself by saying: "Hi, I'm *grep*, I look for patterns in files. Please enter your pattern." After getting the pattern, *grep* prompts for a file name. Then it asks if there are any more file names. Finally, it summarizes what it is going to do and asks if that is correct. While this kind of user interface may be suitable for rank novices, it drives skilled programmers up the wall. What they want is a servant, not a nanny.

10.2.2 Interfaces to Linux

Linux system can be regarded as a kind of pyramid, as illustrated in Fig. 10-1. At the bottom is the hardware, consisting of the CPU, memory, disks, a monitor and keyboard, and other devices. Running on the bare hardware is the operating system. Its function is to control the hardware and provide a system call interface to all the programs. These system calls allow user programs to create and manage processes, files, and other resources.

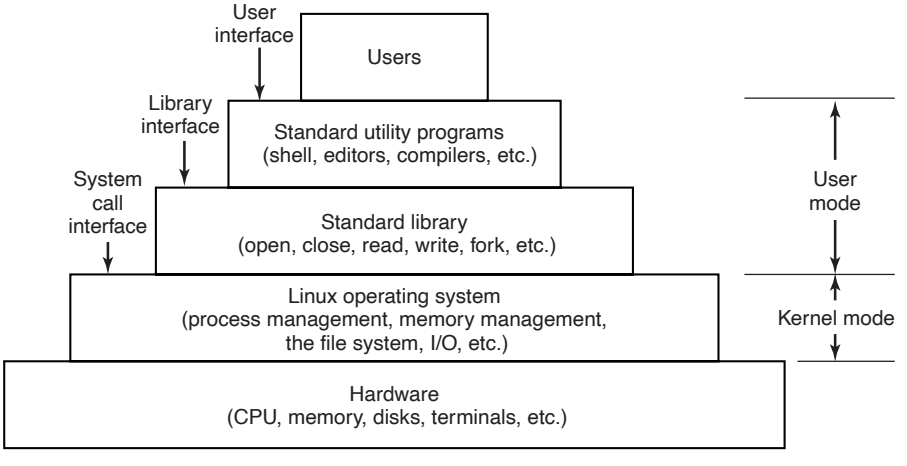

Figure 10-1. The layers in a Linux system.

Programs make system calls by putting the arguments in registers (or sometimes, on the stack), and issuing trap instructions to switch from user mode to kernel mode. Since there is no way to write a trap instruction in C, a library is provided, with one procedure per system call. These procedures are written in assembly language but can be called from C. Each one first puts its arguments in the proper place, then executes the trap instruction. Thus to execute the read system call, a C program can call the read library procedure. As an aside, it is the library interface, and not the system call interface, that is specified by POSIX. In other words, POSIX tells which library procedures a conformant system must supply, what their parameters are, what they must do, and what results they must return. It does not even mention the actual system calls.

In addition to the operating system and system call library, all versions of Linux supply a large number of standard programs, some of which are specified by the POSIX 1003.1-2017 standard, and some of which differ between Linux versions. These include the command processor (shell), compilers, editors, text-processing programs, and file-manipulation utilities. It is these programs that a user at the keyboard invokes. Thus, we can speak of three different interfaces to Linux: the true system call interface, the library interface, and the interface formed by the set of standard utility programs.

Most of the common personal computer distributions of Linux have replaced this keyboard-oriented user interface with a mouse- or a touchscreen-oriented graphical user interface, without changing the operating system itself at all. It is precisely this flexibility that makes Linux so popular and has allowed it to survive numerous changes in the underlying technology so well.

The GUI for Linux is similar to the first GUIs developed for UNIX systems in the 1970s, and popularized by Macintosh and later Windows for PC platforms. The GUI creates a desktop environment, a familiar metaphor with windows, icons, folders, toolbars, and drag-and-drop capabilities. A full desktop environment contains a window manager, which controls the placement and appearance of windows, as well as various applications, and provides a consistent graphical interface. Popular desktop environments for Linux include GNOME (GNU Network Object Model Environment) and KDE (K Desktop Environment).

GUIs on Linux are supported by the X Windowing System, or commonly X11 or just X, which defines communication and display protocols for manipulating windows on bitmap displays for UNIX and UNIX-like systems. The X server is the main component which controls devices such as the keyboard, mouse, and screen and is responsible for redirecting input to or accepting output from client programs. The actual GUI environment is typically built on top of a low-level library, *xlib*, which contains the functionality to interact with the X server. The graphical interface extends the basic functionality of X11 by enriching the window view, providing buttons, menus, icons, and other options. The X server can be started manually, from a command line, but is typically started during the boot process by a display manager, which displays the graphical login screen for the user.

When working on Linux systems through a graphical interface, users may use mouse clicks to run applications or open files, drag and drop to copy files from one location to another, and so on. In addition, users may invoke a terminal emulator program, or *xterm*, which provides them with the basic command-line interface to the operating system. Its description is given in the following section.

10.2.3 The Shell

Although Linux systems have a graphical user interface, most programmers and sophisticated users still prefer a command-line interface, called the **shell**. Often they start one or more shell windows from the graphical user interface and just work in them. The shell command-line interface is much faster to use, more powerful, easily extensible, and does not give the user RSI from having to use a mouse all the time. Below we will briefly describe the bash shell (*bash*). It is heavily based on the original UNIX shell, *Bourne shell* (written by Steve Bourne, then at Bell Labs). Its name is an acronym for *Bourne Again SHell*. Many other shells are also in use (*ksh*, *csh*, etc.), but *bash* is the default shell in most Linux systems.

When the shell starts up, it initializes itself, then types a **prompt** character, often a percent or dollar sign, on the screen and waits for the user to type a command line.

When the user types a command line, the shell extracts the first word from it, where word here means a run of characters delimited by a space or tab. It then assumes this word is the name of a program to be run, searches for this program, and if it finds it, runs the program. The shell then suspends itself until the program terminates, at which time it tries to read the next command. What is important here is simply the observation that the shell is an ordinary user program. All it needs is the ability to read from the keyboard and write to the monitor and the power to execute other programs.

Commands may take arguments, which are passed to the called program as character strings. For example, the command line

```
cp src dest
```

invokes the *cp* program with two arguments, *src* and *dest*. This program interprets the first one to be the name of an existing file. It makes a copy of this file and calls the copy *dest*.

Not all arguments are file names. In

```
head –20 file
```

the first argument, *–20*, tells *head* to print the first 20 lines of *file*, instead of the default number of lines, 10. Arguments that control the operation of a command or specify an optional value are called **flags**, and by convention are indicated with a dash. The dash is required to avoid ambiguity, because the command

```
head 20 file
```

is perfectly legal, and tells *head* to first print the initial 10 lines of a file called *20*, and then print the initial 10 lines of a second file called *file*. Most Linux commands accept multiple flags and arguments.

To make it easy to specify multiple file names, the shell accepts **magic characters**, sometimes called **wild cards**. An asterisk, for example, matches all possible strings, so

```
ls *.c
```

tells *ls* to list all the files whose name ends in *.c*. If files named *x.c*, *y.c*, and *z.c* all exist, the above command is equivalent to typing

```
ls x.c y.c z.c
```

Another wild card is the question mark, which matches any one character. A list of characters inside square brackets selects any of them, so

```
ls [ape]*
```

lists all files beginning with "a", "p", or "e".

A program like the shell does not have to open the terminal (keyboard and monitor) in order to read from it or write to it. Instead, when it (or any other program) starts up, it automatically has access to a file called **standard input** (for reading), a file called **standard output** (for writing normal output), and a file called **standard error** (for writing error messages). Normally, all three default to the terminal, so that reads from standard input come from the keyboard and writes to standard output or standard error go to the screen. Many Linux programs read from standard input and write to standard output as the default. For example,

```
sort
```

invokes the *sort* program, which reads lines from the terminal (until the user types a CTRL-D, to indicate end of file), sorts them alphabetically, and writes the result to the screen.

It is also possible to redirect standard input and standard output, as that is often useful. The syntax for redirecting standard input uses a less-than symbol (<) followed by the input file name. Similarly, standard output is redirected using a greater-than symbol (>). It is permitted to redirect both in the same command. For example, the command

```
sort <in >out
```

causes *sort* to take its input from the file *in* and write its output to the file *out*. Since standard error has not been redirected, any error messages go to the screen. A program that reads its input from standard input, does some processing on it, and writes its output to standard output is called a **filter**.

Consider the following command line consisting of three separate commands separated by semicolons:

```
sort <in >temp; head –30 <temp; rm temp
```

It first runs *sort*, taking the input from *in* and writing the output to *temp*. When that has been completed, the shell runs *head*, telling it to print the first 30 lines of *temp* and print them on standard output, which defaults to the terminal. Finally, the temporary file is removed. It does not go to some special recycling bin. It is gone with the wind, forever.

It frequently occurs that the first program in a command line produces output that is used as input to the next program. In the above example, we used the file *temp* to hold this output. However, Linux provides a simpler construction to do the same thing. In

```
sort <in | head –30
```

the vertical bar, called the **pipe symbol**, says to take the output from *sort* and use it as the input to *head*, eliminating the need for creating, using, and removing the temporary file. A collection of commands connected by pipe symbols, called a **pipeline**, may contain arbitrarily many commands. A four-component pipeline is shown by the following example:

```
grep ter *.t | sort | head –20 | tail –5 >foo
```

Here all the lines containing the string "ter" in all the files ending in *.t* are written to standard output, where they are sorted. The first 20 of these are selected out by *head*, which passes them to *tail*, which writes the last five (i.e., lines 16 to 20 in the sorted list) to *foo*. This is an example of how Linux provides basic building blocks (numerous filters), each of which does one job, along with a mechanism for them to be put together in almost limitless ways.

Linux is a general-purpose multiprogramming system. A single user can run several programs at once, each as a separate process. The shell syntax for running a process in the background is to follow its command with an ampersand. Thus

```
wc –l <a >b &
```

runs the word-count program, *wc*, to count the number of lines (*–l* flag) in its input, *a*, writing the result to *b*, but does it in the background. As soon as the command has been typed, the shell types the prompt and is ready to accept and handle the next command. Pipelines can also be put in the background, for example, by

```
sort <x | head &
```

Multiple pipelines can run in the background simultaneously.

It is also possible to put a list of shell commands in a file and then start a shell with this file as standard input. The (second) shell just processes them in order, the same as it would with commands typed on the keyboard. Files containing shell

commands are called **shell scripts**. Shell scripts may assign values to shell variables and then read them later. They may also have parameters, and use if, for, while, and case constructs. Thus a shell script is really a program written in shell language. The Berkeley C shell is an alternative shell designed to make shell scripts (and the command language in general) look like C programs in many respects. Since the shell is just another user program, other people have written and distributed a variety of other shells. Users are free to choose whatever shells they like.

10.2.4 Linux Utility Programs

The command-line (shell) user interface to Linux consists of a large number of standard utility programs. Roughly speaking, these programs can be divided into six categories, as follows:

1. File and directory manipulation commands.
2. Filters.
3. Program development tools, such as editors and compilers.
4. Text processing.
5. System administzration.
6. Miscellaneous.

The POSIX 1003.1-2017 standard specifies the syntax and semantics of 160 of these, primarily in the first three categories. The idea of standardizing them is to make it possible for anyone to write shell scripts that use these programs and work on all Linux systems.

In addition to these standard utilities, there are many application programs as well, of course, such as Web browsers, media players, image viewers, office suites, games, and so on.

Let us consider some examples of these programs, starting with file and directory manipulation.

```
cp a b
```

copies file *a* to *b*, leaving the original file intact. In contrast,

```
mv a b
```

copies *a* to *b* but removes the original. In effect, it moves the file rather than really making a copy in the usual sense. Several files can be concatenated using *cat*, which reads each of its input files and copies them all to standard output, one after another. Files can be removed by the *rm* command. The *chmod* command allows the owner to change the rights bits to modify access permissions. Directories can

be created with *mkdir* and removed with *rmdir*. To see a list of the files in a directory, *ls* can be used. It has a vast number of flags to control how much detail about each file is shown (e.g., size, owner, group, creation date), to determine the sort order (e.g., alphabetical, by time of last modification, reversed), to specify the layout on the screen, and much more.

We have already seen several filters: *grep* extracts lines containing a given pattern from standard input or one or more input files; *sort* sorts its input and writes it on standard output; *head* extracts the initial lines of its input; *tail* extracts the final lines of its input. Other filters defined by 1003.1 are *cut* and *paste*, which allow columns of text to be cut and pasted into files; *od*, which converts its (usually binary) input to ASCII text, in octal, decimal, or hexadecimal; *tr*, which does character translation (e.g., lowercase to uppercase), and *pr*, which formats output for the printer, including options to include running heads, page numbers, and so on.

Compilers and programming tools include *cc*, which calls the C compiler, and *ar*, which collects library procedures into archive files.

Another important tool is *make*, which is used to maintain large programs whose source code consists of multiple files. Typically, some of these are **header files**, which contain type, variable, macro, and other declarations. Source files often include these using a special *include* directive. This way, two or more source files can share the same declarations. However, if a header file is modified, it is necessary to find all the source files that depend on it and recompile them. The function of *make* is to keep track of which file depends on which header, and similar things, and arrange for all the necessary compilations to occur automatically. Nearly all Linux programs, except some of the very smallest ones, are set up to be compiled with *make*.

A selection of the POSIX utility programs is listed in Fig. 10-2, along with a short description of each. All Linux systems have them and many more.

10.2.5 Kernel Structure

In Fig. 10-1 we saw the overall structure of a Linux system. Now let us zoom in and look more closely at the kernel as a whole before examining the various parts, such as process scheduling and the file system.

The kernel sits directly on the hardware and enables interactions with I/O devices and the memory management unit and controls CPU access to them. At the lowest level, as shown in Fig. 10-3 it contains interrupt handlers, which are the primary way for interacting with devices, and the low-level dispatching mechanism. This dispatching occurs when an interrupt happens. The low-level code here stops the running process, saves its state in the kernel process structures, and starts the appropriate driver. Process dispatching also happens when the kernel completes some operations and it is time to start up a user process again. The dispatching code is in assembler and is quite distinct from scheduling.

Program	Typical use
cat	Concatenate multiple files to standard output
chmod	Change file protection mode
cp	Copy one or more files
cut	Cut columns of text from a file
grep	Search a file for some pattern
head	Extract the first lines of a file
ls	List directory
make	Compile files to build a binary
mkdir	Make a directory
od	Octal dump a file
paste	Paste columns of text into a file
pr	Format a file for printing
ps	List running processes
rm	Remove one or more files
rmdir	Remove a directory
sort	Sort a file of lines alphabetically
tail	Extract the last lines of a file
tr	Translate between character sets

Figure 10-2. A few of the common Linux utility programs required by POSIX.

Next, we divide the various kernel subsystems into three main components. The I/O component in Fig. 10-3 contains all kernel pieces responsible for interacting with devices and performing network and storage I/O operations. At the highest level, the I/O operations are all integrated under a **VFS** (**Virtual File System**) layer. That is, at the top level, performing a read operation on a file, whether it is in memory or on disk, is the same as performing a read operation to retrieve a character from a terminal input. At the lowest level, all I/O operations pass through some device driver. All Linux drivers are classified as either character-device drivers or block-device drivers, the main difference being that seeks and random accesses are allowed on block devices and not on character devices. Technically, network devices are really character devices, but they are handled somewhat differently, so that it is probably clearer to separate them, as has been done in the figure.

Above the device-driver level, the kernel code is different for each device type. Character devices may be used in two different ways. Some programs, such as visual editors like *vi* and *emacs*, want every keystroke as it is hit. Raw terminal (tty) I/O makes this possible. Other software, such as the shell, is line oriented, allowing users to edit the whole line before hitting ENTER to send it to the program. In this case, the character stream from the terminal device is passed through a so-called line discipline, and appropriate formatting is applied.

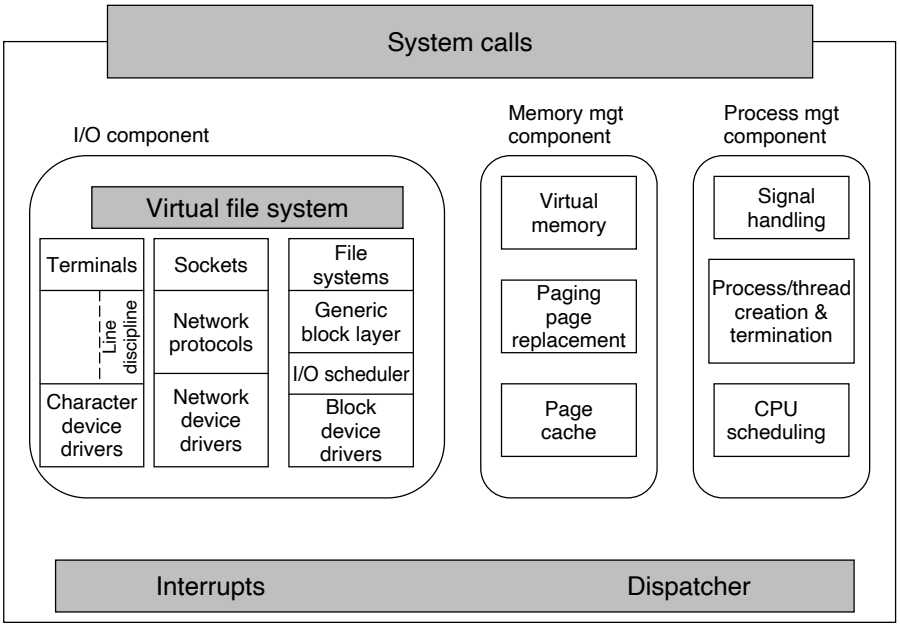

Figure 10-3. Structure of the Linux kernel.

Networking software is often modular, with different devices and protocols supported. The layer above the network drivers handles a kind of routing function, making sure that the right packet goes to the right device or protocol handler. Most Linux systems contain the full functionality of a hardware router within the kernel, although the performance is less than that of a hardware router. Above the router code is the actual protocol stack, including IP and TCP, but also many additional protocols. Overlaying all the network is the socket interface, which allows programs to create sockets for particular networks and protocols, getting back a file descriptor for each socket to use later.

On top of the disk drivers is the I/O scheduler, which is responsible for ordering and issuing disk-operation requests in a way that tries to conserve wasteful disk head movement or to meet some other system policy.

At the very top of the block-device column are the file systems. Linux may, and in fact does, have multiple file systems coexisting concurrently. In order to hide the gruesome architectural differences of various hardware devices from the file system implementation, a generic block-device layer provides an abstraction used by all file systems.

In the right half of Fig. 10-3 are the other two key components of the Linux kernel. These two are responsible for the memory and process management tasks. Memory-management tasks include maintaining the virtual to physical-memory

page mappings, maintaining a cache of recently accessed pages and implementing a good page-replacement policy, and on-demand bringing in new pages of needed code and data into memory.

The key responsibility of the process-management component is the creation and termination of processes. It also includes the process scheduler, which chooses which process or, rather, thread to run next. As we shall see in the next section, the Linux kernel treats both processes and threads simply as executable entities, and will schedule them based on a global scheduling policy. Finally, code for signal handling also belongs to this component.

While the three components are represented separately in the figure, they are highly interdependent. File systems typically access files through the block devices. However, in order to hide the large latencies of disk accesses, files are copied into the page cache in main memory. Some files may even be dynamically created and may have only an in-memory representation, such as files providing some run-time resource usage information. In addition, the virtual memory system may rely on a disk partition or in-file swap area to back up parts of the main memory when it needs to free up certain pages, and therefore relies on the I/O component. Numerous other interdependencies exist.

In addition to the static in-kernel components, Linux supports dynamically loadable modules. These modules can be used to add or replace the default device drivers, file system, networking, or other kernel codes. The modules are not shown in Fig. 10-3.

Finally, at the very top is the system call interface into the kernel. All system calls come here, causing a trap which switches the execution from user mode into protected kernel mode and passes control to one of the kernel components described earlier.

10.3 PROCESSES IN LINUX

In the previous sections, we started out by looking at Linux as viewed from the keyboard, that is, what the user sees in an *xterm* window. We gave examples of shell commands and utility programs that are frequently used. We ended with a brief overview of the system structure. Now it is time to dig deeply into the kernel and look more closely at the basic concepts Linux supports, namely, processes, memory, the file system, and input/output. These notions are important because the system calls—the interface to the operating system itself—manipulate them. For example, system calls exist to create processes and threads, allocate memory, open files, and do I/O.

Unfortunately, with so many distributions of Linux in existence (and old versions of the kernel still widely used), there are some differences between them. In this chapter, we will emphasize the features common to all of them rather than focus on any one specific version. Thus in certain sections (especially implementation sections), the discussion may not apply equally to every version.

10.3.1 Fundamental Concepts

The main active entities in a Linux system are the processes. Linux processes are very similar to the classical sequential processes that we studied in Chap. 2. Each process runs a single program and initially has a single thread of control. In other words, it has one program counter, which keeps track of the next instruction to be executed. Linux allows a process to create additional threads once it starts.

Linux is a multiprogramming system, so multiple, independent processes may be running at the same time. Furthermore, each user may have several active processes at once, so on a large system, there may be hundreds or even thousands of processes running. In fact, on most single-user workstations, even when the user is absent, dozens of background processes, called **daemons**, are running. These are started by a shell script when the system is booted. ("Daemon" is a variant spelling of "demon," which is a self-employed evil spirit.)

A typical daemon is the *cron daemon*. It wakes up once a minute to check if there is any work for it to do. If so, it does the work. Then it goes back to sleep until it is time for the next check.

This daemon is needed because it is possible in Linux to schedule activities minutes, hours, days, or even months in the future. For example, suppose a user has a dentist appointment at 3 o'clock next Tuesday. He can make an entry in the cron daemon's database telling the daemon to beep at him at, say, 2:30. When the appointed day and time arrives, the cron daemon sees that it has work to do, and starts up the beeping program as a new process.

The cron daemon is also used to start up periodic activities, such as making daily disk backups at 4 A.M., or reminding forgetful users every year on October 31 to stock up on trick-or-treat goodies for Halloween. Other daemons handle incoming and outgoing electronic mail, manage the line printer queue, check if there are enough free pages in memory, and so forth. Daemons are straightforward to implement in Linux because each one is a separate process, independent of all other processes.

Processes are created in Linux in an especially simple manner. The fork system call creates an exact copy of the original process. The forking process is called the **parent process**. The new process is called the **child process**. The parent and child each have their own, private memory images. If the parent subsequently changes any of its variables, the changes are not visible to the child, and vice versa.

Open files are shared between parent and child. That is, if a certain file was open in the parent before the fork, it will continue to be open in both the parent and the child afterward. Changes made to the file by either one will be visible to the other. This behavior is only reasonable, because these changes are also visible to any unrelated process that opens the file.

The fact that the memory images, variables, registers, and everything else are identical in the parent and child leads to a small difficulty: How do the processes know which one should run the parent code and which one should run the child

code? The secret is that the fork system call returns a 0 to the child and a nonzero value, the child's **PID** (**Process Identifier**), to the parent. Both processes normally check the return value and act accordingly, as shown in Fig. 10-4.

```
pid = fork( );                  /* if the fork succeeds, pid > 0 in the parent */
if (pid < 0) {
     handle_error( );           /* fork failed (e.g., memory or some table is full) */
} else if (pid > 0) {
                                /* parent code goes here. /*/
} else {
                                /* child code goes here. /*/
}
```

Figure 10-4. Process creation in Linux.

Processes are named by their PIDs. When a process is created, the parent is given the child's PID, as mentioned above. If the child wants to know its own PID, there is a system call, getpid, that provides it. PIDs are used in a variety of ways. For example, when a child terminates, the parent is given the PID of the child that just finished. This can be important because a parent may have many children. Since children may also have children, an original process can build up an entire tree of children, grandchildren, and further descendants.

Processes in Linux can communicate with each other using a form of message passing. It is possible to create a channel between two processes into which one process can write a stream of bytes for the other to read. These channels are called **pipes**. Synchronization is possible because when a process tries to read from an empty pipe it is blocked until data are available.

Shell pipelines are implemented with pipes. When the shell sees a line like

```
sort <f | head
```

it creates two processes, *sort* and *head*, and sets up a pipe between them in such a way that *sort*'s standard output is connected to *head*'s standard input. In this way, all the data that *sort* writes go directly to *head*, instead of going to a file. If the pipe fills, the system stops running *sort* until *head* has removed some data from it.

Processes can also communicate in another way besides pipes: software interrupts. A process can send what is called a **signal** to another process. Processes can tell the system what they want to happen when an incoming signal arrives. The choices available are to ignore it, to catch it, or to let the signal kill the process. Terminating the process is the default for most signals. If a process elects to catch signals sent to it, it must specify a signal-handling procedure. When a signal arrives, control will abruptly switch to the handler. When the handler is finished and returns, control goes back to where it came from, analogous to hardware I/O interrupts. A process can send signals only to members of its **process group**, which consists of its parent (and further ancestors), siblings, and children (and

further descendants). A process may also send a signal to all members of its process group with a single system call.

Signals are also used for other purposes. For example, if a process is doing floating-point arithmetic, and inadvertently divides by 0 (something that mathematicians tend to frown upon), it gets a SIGFPE (floating-point exception) signal. Some of the signals that are required by POSIX are listed in Fig. 10-5. Many Linux systems have additional signals as well, but programs using them may not be portable to other versions of Linux and UNIX in general.

Signal	Cause
SIGABRT	Sent to abort a process and force a core dump
SIGALRM	The alarm clock has gone off
SIGFPE	A floating-point error has occurred (e.g., division by 0)
SIGHUP	The telecommunications connection was lost
SIGILL	The process has tried to execute an illegal instruction
SIGQUIT	The user has hit the key requesting a core dump
SIGKILL	Sent to kill a process (cannot be caught or ignored)
SIGPIPE	The process has written to a pipe which has no readers
SIGSEGV	The process has referenced an invalid memory address
SIGTERM	Used to request that a process terminate gracefully
SIGUSR1	Available for application-defined purposes
SIGUSR2	Available for application-defined purposes

Figure 10-5. Some of the signals required by POSIX.

10.3.2 Process-Management System Calls in Linux

Let us now look at the Linux system calls dealing with process management. The main ones are listed in Fig. 10-6. Fork is a good place to start the discussion. The fork system call, supported also by other traditional UNIX systems, is the main way to create a new process in Linux systems. (We will discuss another alternative in the following section.) It creates an exact duplicate of the original process, including all the file descriptors, registers, and everything else. After the fork, the original process and the copy (the parent and child) go their separate ways. All the variables have identical values at the time of the fork, but since the entire parent address space is copied to create the child, subsequent changes in one of them do not affect the other. The fork call returns a value, which is zero in the child, and equal to the child's PID in the parent. Using the returned PID, the two processes can see which is the parent and which is the child.

In most cases, after a fork, the child will need to execute different code from the parent. Consider the case of the shell. It reads a command from the terminal, forks off a child process, waits for the child to execute the command, and then

System call	Description
pid = fork( )	Create a child process identical to the parent
pid = waitpid(pid, &statloc, opts)	Wait for a child to terminate
s = execve(name, argv, envp)	Replace a process' core image
exit(status)	Terminate process execution and return status
s = sigaction(sig, &act, &oldact)	Define action to take on signals
s = sigreturn(&context)	Return from a signal
s = sigprocmask(how, &set, &old)	Examine or change the signal mask
s = sigpending(set)	Get the set of blocked signals
s = sigsuspend(sigmask)	Replace the signal mask and suspend the process
s = kill(pid, sig)	Send a signal to a process
residual = alarm(seconds)	Set the alarm clock
s = pause( )	Suspend the caller until the next signal

Figure 10-6. Some system calls relating to processes. The return code *s* is −1 if an error has occurred, *pid* is a process ID, and *residual* is the remaining time in the previous alarm. The parameters are what the names suggest.

reads the next command when the child terminates. To wait for the child to finish, the parent executes a waitpid system call, which just waits until the child terminates (any child if more than one exists). Waitpid has three parameters. The first one allows the caller to wait for a specific child. If it is −1, any old child (i.e., the first child to terminate) will do. The second parameter is the address of a variable that will be set to the child's exit status (normal or abnormal termination and exit value). This allows the parent to know the fate of its child. The third parameter determines whether the caller blocks or returns if no child is already terminated.

In the case of the shell, the child process must execute the command typed by the user. It does this by using the exec system call, which causes its entire core image to be replaced by the file named in its first parameter. A highly simplified shell illustrating the use of fork, waitpid, and exec is shown in Fig. 10-7.

In the most general case, exec has three parameters: the name of the file to be executed, a pointer to the argument array, and a pointer to the environment array. These will be described shortly. Various library procedures, such as *execl*, *execv*, *execle*, and *execve*, are provided to allow the parameters to be omitted or specified in various ways. All of these procedures invoke the same underlying system call. Although the system call is exec, there is no library procedure with this name; one of the others must be used.

Let us consider the case of a command typed to the shell, such as

```
cp file1 file2
```

used to copy *file1* to *file2*. After the shell has forked, the child locates and executes the file *cp* and passes it information about the files to be copied.

```
while (TRUE) {                                  /* repeat forever /*/
      type_prompt( );                           /* display prompt on the screen */
      read_command(command, params);            /* read input line from keyboard */

      pid = fork( );                            /* fork off a child process */
      if (pid < 0) {
            printf("Unable to fork0);           /* error condition */
            continue;                           /* repeat the loop */
      }

      if (pid != 0) {
            waitpid (-1, &status, 0);           /* parent waits for child */
      } else {
            execve(command, params, 0);         /* child does the work */
      }
}
```

Figure 10-7. A highly simplified shell.

The main program of *cp* (and many other programs) contains the function declaration

```
main(argc, argv, envp)
```

where *argc* is a count of the number of items on the command line, including the program name. For the example above, *argc* is 3.

The second parameter, *argv*, is a pointer to an array. Element *i* of that array is a pointer to the *i*th string on the command line. In our example, *argv*[0] would point to the two-character string "cp". Similarly, *argv*[1] would point to the five-character string "file1" and *argv*[2] would point to the five-character string "file2".

The third parameter of *main*, *envp*, is a pointer to the environment, an array of strings containing assignments of the form *name = value* used to pass information such as the terminal type and home directory name to a program. In Fig. 10-7, no environment is passed to the child, so that the third parameter of *execve* is a zero.

If exec seems complicated, do not despair; it is the most complex system call. All the rest are much simpler. As an example of a simple one, consider exit, which processes should use when they are finished executing. It has one parameter, the exit status (0 to 255), which is returned to the parent in the variable *status* of the waitpid system call. The low-order byte of *status* contains the termination status, with 0 being normal termination and the other values being various error conditions. The high-order byte contains the child's exit status (0 to 255), as specified in the child's call to exit. For example, if a parent process executes the statement

```
n = waitpid(-1, &status, 0);
```

it will be suspended until some child process terminates. If the child exits with, say, 4 as the parameter to *exit*, the parent will be awakened with *n* set to the child's

PID and *status* set to 0x0400 (0x as a prefix means hexadecimal in C). The low-order byte of *status* relates to signals; the next one is the value the child returned in its call to exit.

If a process exits and its parent has not yet waited for it, the process enters a kind of suspended animation called the **zombie state**—the living dead. When the parent finally waits for it, the process terminates.

Several system calls relate to signals, which are used in a variety of ways. For example, if a user accidentally tells a text editor to display the entire contents of a very long file, and then realizes the error, some way is needed to interrupt the editor. The usual choice is for the user to hit some special key (e.g., DEL or CTRL-C), which sends a signal to the editor. The editor catches the signal and stops.

To announce its willingness to catch this (or any other) signal, the process can use the sigaction system call. The first parameter is the signal to be caught (see Fig. 10-5). The second is a pointer to a structure giving a pointer to the signal-handling procedure, as well as some other bits and flags. The third one points to a structure where the system returns information about signal handling currently in effect, in case it must be restored later.

The signal handler may run for as long as it wants to. In practice, though, signal handlers are usually fairly short. When the signal-handling procedure is done, it returns to the point from which it was interrupted.

The sigaction system call can also be used to cause a signal to be ignored, or to restore the default action, which is killing the process.

Hitting the DEL or CTRL key is not the only way to send a signal. The kill system call allows a process to signal another related process. The choice of the name "kill" for this system call is not an especially good one, since most processes send signals to other ones with the intention that they be caught. However, a signal that is not caught, does, indeed, kill the recipient.

For many real-time applications, a process needs to be interrupted after a specific time interval to do something, such as to retransmit a potentially lost packet over an unreliable communication line. To handle this situation, the alarm system call has been provided. The parameter specifies an interval, in seconds, after which a SIGALRM signal is sent to the process. A process may have only one alarm outstanding at any instant. If an alarm call is made with a parameter of 10 seconds, and then 3 seconds later another alarm call is made with a parameter of 20 seconds, only one signal will be generated, 20 seconds after the second call. The first signal is canceled by the second call to alarm. If the parameter to alarm is zero, any pending alarm signal is canceled. If an alarm signal is not caught, the default action is taken and the signaled process is killed. Technically, alarm signals may be ignored, but that is a pointless thing to do. Why would a program ask to be signaled later on and then ignore the signal?

It sometimes occurs that a process has nothing to do until a signal arrives. For example, consider a computer-aided instruction program that is testing reading speed and comprehension. It displays some text on the screen and then calls alarm

to signal it after 30 seconds. While the student is reading the text, the program has nothing to do. It could sit in a tight loop doing nothing, but that would waste CPU time that a background process or other user might need. A better solution is to use the pause system call, which tells Linux to suspend the process until the next signal arrives. Woe be it to the program that calls pause with no alarm pending.

10.3.3 Implementation of Processes and Threads in Linux

A process in Linux is like an iceberg: you only see the part above the water, but there is also an important part underneath. Every process has a user part that runs the user program. However, when one of its threads makes a system call, it traps to kernel mode and begins running in kernel context, with a different memory map and full access to all machine resources. It is still the same thread, but now with more power and also its own kernel mode stack and kernel mode program counter. These are important because a system call can block partway through, for example, waiting for a disk operation to complete. The program counter and registers are then saved so the thread can be restarted in kernel mode later.

The Linux kernel internally represents processes as **tasks**, via the structure *task_struct*. Unlike other OS approaches (which make a distinction between a process, lightweight process, and thread), Linux uses the task structure to represent any execution context. Therefore, a single-threaded process will be represented with one task structure and a multithreaded process will have one task structure for each of the user-level threads. Finally, the kernel itself is multithreaded, and has kernel-level threads which are not associated with any user process and are executing kernel code. We will return to the treatment of multithreaded processes (and threads in general) later in this section.

For each process, a process descriptor of type *task_ struct* is resident in memory at all times. It contains vital information needed for the kernel's management of all processes, including scheduling parameters, lists of open-file descriptors, and so on. The process descriptor along with memory for the kernel-mode stack for the process are created upon process creation.

For compatibility with other UNIX systems, Linux identifies processes via the PID. The kernel organizes all processes in a doubly linked list of task structures. In addition to accessing process descriptors by traversing the linked lists, the PID can be mapped to the address of the task structure, and the process information can be accessed immediately.

The task structure contains a variety of fields. Some of these fields contain pointers to other data structures or segments, such as those containing information about open files. Some of these segments are related to the user-level structure of the process, which is not of interest when the user process is not runnable. Therefore, these may be swapped or paged out, in order not to waste memory on information that is not needed. For example, although it is possible for a process to be sent a signal while it is swapped out, it is not possible for it to read a file. For this

reason, information about signals must be in memory all the time, even when the process is not present in memory. On the other hand, information about file descriptors can be kept in the user structure and brought in only when the process is in memory and runnable.

The information in the process descriptor falls into a number of broad categories that can be roughly described as follows:

1. **Scheduling parameters**. Process priority, amount of CPU time consumed recently, amount of time spent sleeping recently. Together, these are used to determine which process to run next.
2. **Memory image**. Pointers to the text, data, and stack segments, or page tables. If the text segment is shared, the text pointer points to the shared text table. When the process is not in memory, information about how to find its parts on disk is here too.
3. **Signals**. Masks showing which signals are being ignored, which are being caught, which are being temporarily blocked, and which are in the process of being delivered.
4. **Machine registers**. When a trap to the kernel occurs, the machine registers (including the floating-point ones, if used) are saved here.
5. **System call state**. Information about the current system call, including the parameters, and results.
6. **File descriptor table**. When a system call involving a file descriptor is invoked, the file descriptor is used as an index into this table to locate the in-core data structure (i-node) corresponding to this file.
7. **Accounting**. Pointer to a table that keeps track of the user and system CPU time used by the process. Some systems also maintain limits here on the amount of CPU time a process may use, the maximum size of its stack, the number of page frames it may consume, and other items.
8. **Kernel stack**. A fixed stack for use by the kernel part of the process.
9. **Miscellaneous**. Current process state, event being waited for, if any, time until alarm clock goes off, PID, PID of the parent process, and user and group identification.

Keeping this information in mind, it is now easy to explain how processes are created in Linux. The mechanism for creating a new process is actually fairly straightforward. A new process descriptor and user area are created for the child process and filled in largely from the parent. The child is given a unique PID not used by any other process, its memory map is set up, and it is given shared access to its parent's files. Then its registers are set up and it is ready to run.

When a fork system call is executed, the calling process traps to the kernel and creates a task structure and few other accompanying data structures, such as the kernel-mode stack and a *thread_info* structure. This structure is allocated at a fixed offset from the process' end-of-stack, and contains few process parameters, along with the address of the process descriptor. By storing the process descriptor's address at a fixed location, Linux needs only few efficient operations to locate the task structure for a running process.

The majority of the process-descriptor contents are filled out based on the parent's descriptor values. Linux then looks for an available PID, that is, not one currently in use by any process, and updates the PID hash-table entry to point to the new task structure. In case of collisions in the hash table, process descriptors may be chained. It also sets the fields in the *task_struct* to point to the corresponding previous/next process on the task array.

In principle, it should now allocate memory for the child's data and stack segments, and to make exact copies of the parent's segments, since the semantics of fork say that no memory is shared between parent and child. The text segment may be either copied or shared since it is read only. At this point, the child is ready to run.

However, copying memory is expensive, so all modern Linux systems cheat. They give the child its own page tables, but have them point to the parent's pages, only marked read only. Whenever either process (the child or the parent) tries to write on a page, it gets a protection fault. The kernel sees this and then allocates a new copy of the page to the faulting process and marks it read/write. In this way, only pages that are actually written have to be copied. This mechanism is called **COW** (**Copy On Write**). It has the additional benefit of not requiring two copies of the program in memory, thus saving RAM.

After the child process starts running, the code running there (a copy of the shell in our example) does an exec system call giving the command name as a parameter. The kernel now finds and verifies the executable file, copies the arguments and environment strings to the kernel, and releases the old address space and its page tables.

Now the new address space must be created and filled in. If the system supports mapped files, as Linux and virtually all other UNIX-based systems do, the new page tables are set up to indicate that no pages are in memory, except perhaps one stack page, but that the address space is backed by the executable file on disk. When the new process starts running, as soon as it touches memory to fetch the first instruction, it will immediately get a page fault, which will cause the first page of code to be paged in from the executable file. In this way, nothing has to be loaded in advance, so programs can start quickly and fault in just those pages they need and no more. (This strategy is really just demand paging in its most pure form, as we discussed in Chap. 3.) Finally, the arguments and environment strings are copied to the new stack, the signals are reset, and the registers are initialized to all zeros. At this point, the new command can start running.

Figure 10-8 illustrates the steps described above through the following example. After the user types the command, ls, the shell creates a new process by forking off a clone of itself. The new shell then calls exec to overlay its memory with the contents of the executable file *ls*. After that, *ls* can start.

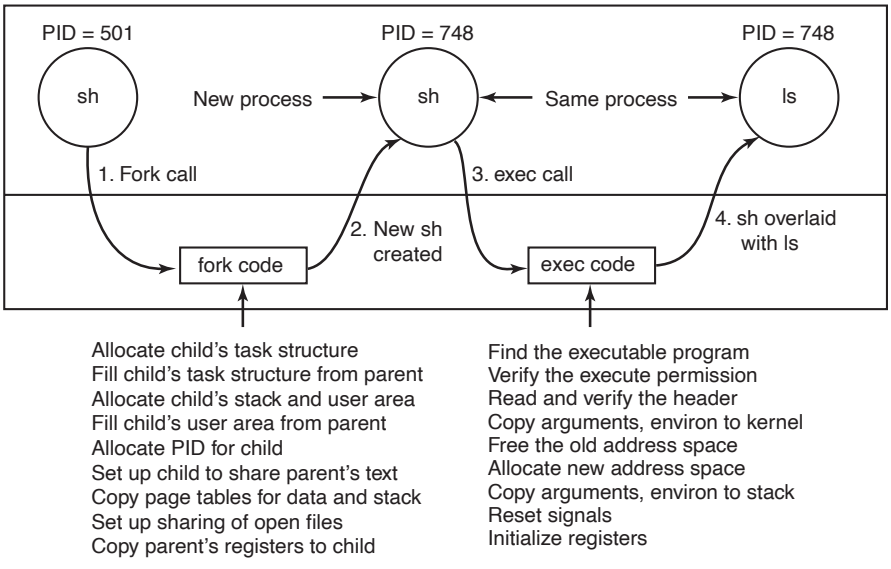

Figure 10-8. The steps in executing the command *ls* typed to the shell.

Threads in Linux

We discussed threads in a general way in Chap. 2. Here we will focus on kernel threads in Linux, particularly on the differences among the Linux thread model and other UNIX systems. In order to better understand the unique capabilities provided by the Linux model, we start with a discussion of some of the challenging decisions present in multithreaded systems.

The main issue in introducing threads is maintaining the correct traditional UNIX semantics. First consider fork. Suppose that a process with multiple (kernel) threads does a fork system call. Should all the other threads be created in the new process? For the moment, let us answer that question with yes. Suppose that one of the other threads was blocked reading from the keyboard. Should the corresponding thread in the new process also be blocked reading from the keyboard? If so, which one gets the next line typed? If not, what should that thread be doing in the new process?

The same problem holds for many other things threads can do. In a single-threaded process, the problem does not arise because the one and only thread

cannot be blocked when calling fork. Now consider the case that the other threads are not created in the child process. Suppose that one of the not-created threads holds a mutex that the one-and-only thread in the new process tries to acquire after doing the fork. The mutex will never be released and the one thread will hang forever. Numerous other problems exist, too. There is no simple solution.

File I/O is another problem area. Suppose that one thread is blocked reading from a file and another thread closes the file or does an lseek to change the current file pointer. What happens next? Who knows?

Signal handling is another thorny issue. Should signals be directed at a specific thread or just at the process? A SIGFPE (floating-point exception) should probably be caught by the thread that caused it. What if it does not catch it? Should just that thread be killed, or all threads? Now consider the SIGINT signal, generated by the user at the keyboard. Which thread should catch that? Should all threads share a common set of signal masks? All solutions to these and other problems usually cause something to break somewhere. Getting the semantics of threads right (not to mention the code) is a nontrivial business.

Linux supports kernel threads in an interesting way that is worth looking at. The implementation is based on ideas from 4.4BSD, but kernel threads were not enabled in that distribution because Berkeley ran out of money before the C library could be rewritten to solve the problems discussed earlier.

Historically, processes were resource containers and threads were the units of execution. A process contained one or more threads that shared the address space, open files, signal handlers, alarms, and everything else. Everything was clear and simple as described above.

In 2000, Linux introduced a powerful new system call, clone, that blurred the distinction between processes and threads and possibly even inverted the primacy of the two concepts. Clone is not present in any other version of UNIX. Classically, when a new thread was created, the original thread(s) and the new one shared everything but their registers. In particular, file descriptors for open files, signal handlers, alarms, and other global properties were per process, not per thread. What clone did was make it possible for each of these aspects and others to be process specific or thread specific. It is called as follows:

```
pid = clone(function, stack_ptr, sharing_flags, arg);
```

The call creates a new thread, either in the current process or in a brand new process, depending on *sharing_flags*. If the new thread is in the current process, it shares the address space with the existing threads, and every subsequent write to any byte in the address space by any thread is immediately visible to all the other threads in the process. On the other hand, if the address space is not shared, then the new thread gets an exact copy of the address space, but subsequent writes by the new thread are not visible to the old ones. These semantics are the same as POSIX. Clone generalizes fork while preserving legacy semantics where needed.

In both cases, the new thread begins executing at *function*, which is called with *arg* as its only parameter. Also in both cases, the new thread gets its own private stack, with the stack pointer initialized to *stack_ptr*.

The *sharing_flags* parameter is a bitmap that allows a finer grain of sharing than traditional UNIX systems. Each of the bits can be set independently of the other ones, and each of them determines whether the new thread copies some data structure or shares it with the calling thread. Figure 10-9 shows some of the items that can be shared or copied according to bits in *sharing_flags*.

Flag	Meaning when set	Meaning when cleared
CLONE_VM	Create a new thread	Create a new process
CLONE_FS	Share umask, root, and working dirs	Do not share them
CLONE_FILES	Share the file descriptors	Copy the file descriptors
CLONE_SIGHAND	Share the signal handler table	Copy the table
CLONE_PARENT	New thread has same parent as caller	New thread's parent is caller

Figure 10-9. Bits in the *sharing_flags* bitmap.

The *CLONE_VM* bit determines whether the virtual memory (i.e., address space) is shared with the old threads or copied. If it is set, the new thread just moves in with the existing ones, so the clone call effectively creates a new thread in an existing process. If the bit is cleared, the new thread gets its own private address space. Having its own address space means that the effect of its STORE instructions is not visible to the existing threads. This behavior is similar to fork, except as noted below. Creating a new address space is effectively the definition of a new process.

The *CLONE_FS* bit controls sharing of the root and working directories and of the umask flag. Even if the new thread has its own address space, if this bit is set, the old and new threads share working directories. This means that a call to chdir by one thread changes the working directory of the other thread, even though the other thread may have its own address space. In UNIX, a call to chdir by a thread always changes the working directory for other threads in its process, but never for threads in another process. Thus this bit enables a kind of sharing not possible in traditional UNIX versions.

The *CLONE_FILES* bit is analogous to the *CLONE_FS* bit. If set, the new thread shares its file descriptors with the old ones, so calls to lseek by one thread are visible to the other ones, again as normally holds for threads within the same process but not for threads in different processes. Similarly, *CLONE_SIGHAND* enables or disables the sharing of the signal handler table between the old and new threads. If the table is shared, even among threads in different address spaces, then changing a handler in one thread affects the handlers in the others.

Finally, every process has a parent. The *CLONE_PARENT* bit controls who the parent of the new thread is. It can either be the same as the calling thread (in

which case the new thread is a sibling of the caller) or it can be the calling thread itself, in which case the new thread is a child of the caller. There are a few other bits that control other items, but they are less important.

This fine-grained sharing is possible because Linux maintains separate data structures for the various items listed in Sec. 10.3.3 (scheduling parameters, memory image, and so on). The task structure just points to these data structures, so it is easy to make a new task structure for each cloned thread and have it point either to the old thread's scheduling, memory, and other data structures or to copies of them. The fact that such fine-grained sharing is possible does not mean that it is useful, however, especially since traditional UNIX versions do not offer this functionality. A Linux program that takes advantage of it is then no longer portable to UNIX.

The Linux thread model raises another difficulty. UNIX systems associate a single PID with a process, independent of whether it is single- or multithreaded. In order to be compatible with other UNIX systems, Linux distinguishes between a process identifier (PID) and a task identifier (TID). Both fields are stored in the task structure. When clone is used to create a new process that shares nothing with its creator, PID is set to a new value; otherwise, the task receives a new TID, but inherits the PID. In this manner, all threads in a process will receive the same PID as the first thread in the process.

10.3.4 Scheduling in Linux

We will now look at the Linux scheduling algorithm. To start with, Linux threads are kernel threads, so scheduling is based on threads, not processes.

Linux distinguishes the following classes of threads for scheduling purposes:

1. Real-time FIFO.
2. Real-time round robin.
3. Sporadic.
4. Timesharing.

Real-time FIFO threads are the highest priority and are not preemptable except by a newly readied real-time FIFO thread with even higher priority. Real-time round-robin threads are the same as real-time FIFO threads except that they have time quanta associated with them, and are preemptable by the clock. If multiple real-time round-robin threads are ready, each one is run for its quantum, after which it goes to the end of the list of real-time round-robin threads. Neither of these classes is actually real time in any sense. Deadlines cannot be specified and guarantees are not given. The sporadic scheduling class is used for sporadic or aperiodic threads, and makes it possible to limit their execution time within a period, so as not to jeopardize other real-time threads. These classes are simply higher priority than

threads in the standard timesharing class. The reason Linux calls them real time is that Linux is conformant to the P1003.4 standard ("real-time" extensions to UNIX) which uses those names. The real-time threads are internally represented with priority levels from 0 to 99, 0 being the highest and 99 the lowest real-time priority level.

The conventional, non-real-time threads form a separate class and are scheduled by a separate algorithm so they do not compete with the real-time threads. Internally, these threads are associated with priority levels from 100 to 139, that is, Linux internally distinguishes among 140 priority levels (for real-time and non-real-time tasks). As for the real-time round-robin threads, Linux allocates CPU time to the non-real-time tasks based on their requirements and their priority levels.

In Linux, time is measured as the number of clock ticks. In older Linux versions, the clock ran at 1000 Hz and each tick was 1 ms, called a **jiffy**. In newer versions, the tick frequency can be configured to 500, 250 or even 1 Hz. In order to avoid wasting CPU cycles for servicing the timer interrupt, the kernel can even be configured in "tickless" mode. This is useful when there is only one process running in the system, or when the CPU is idle and needs to go into power-saving mode. Finally, on newer systems, **high-resolution timers** allow the kernel to keep track of time in sub-jiffy granularity.

Like most UNIX systems, Linux associates a nice value with each thread. The default is 0, but this can be changed using the nice(value) system call, where value ranges from −20 to +19. This value determines the static priority of each thread. A user computing π to a billion places in the background might put this call in his program to be nice to the other users. Only the system administrator may ask for *better* than normal service (meaning values from −20 to −1). Deducing the reason for this rule is left as an exercise for the reader.

Next, we will describe in more detail two of the Linux scheduling algorithms. Their internals are closely related to the design of the **runqueue**, a key data structure used by the scheduler to track all runnable tasks in the system and select the next one to run. A runqueue is associated with each CPU in the system.

Historically, a popular Linux scheduler was the Linux **O(1) scheduler**. It received its name because it was able to perform task-management operations, such as selecting a task or enqueueing a task on the runqueue, in constant time, independent of the total number of tasks in the system. In the O(1) scheduler, the runqueue is organized in two arrays, called *active* and *expired*. As depicted in Fig. 10-10(a), each of these is an array of 140 list heads, each corresponding to a different priority. Each list head points to a doubly linked list of processes at a given priority. The basic operation of the scheduler can be described as follows.

The scheduler selects a task from the highest-priority list in the active array. If that task's timeslice (quantum) expires, it is moved to the expired list (potentially at a different priority level). If the task blocks, for instance to wait on an I/O event, before its timeslice expires, once the event occurs and its execution can resume, it is placed back on the original active array, and its timeslice is decremented to

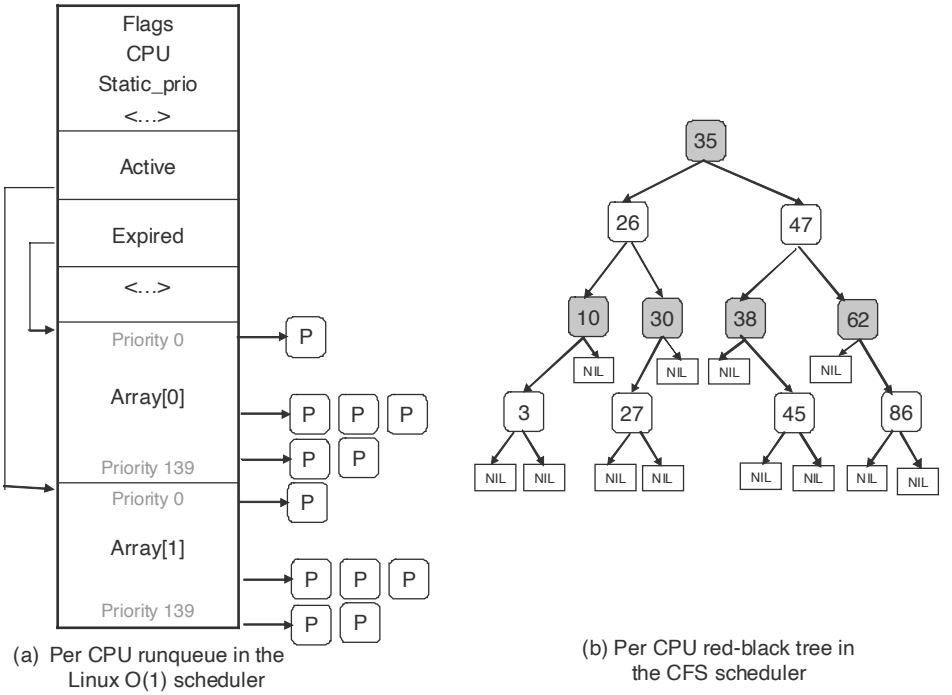

Figure 10-10. Illustration of Linux runqueue data structures for (a) the Linux O(1) scheduler, and (b) the Completely Fair Scheduler.

reflect the CPU time it already used. Once its timeslice is fully exhausted, it, too, will be placed on the expired array. When there are no more tasks in the active array, the scheduler simply swaps the pointers, so the expired arrays now become active, and vice versa. This method ensures that low-priority tasks will not starve (except when real-time FIFO threads completely hog the CPU, which is unlikely).

Here, different priority levels are assigned different timeslice values, with higher quanta assigned to higher-priority processes. For instance, tasks running at priority level 100 will receive time quanta of 800 msec, whereas tasks at priority level of 139 will receive 5 msec.

The idea here is to get processes out of the kernel fast. If a process is trying to read a disk file, making it wait a second between `read` calls will slow it down enormously. It is far better to let it run immediately after each request is completed, so that it can make the next one quickly. Similarly, if a process was blocked waiting for keyboard input, it is clearly an interactive process, and as such should be given a high priority as soon as it is ready in order to ensure that interactive processes get good service. In this light, CPU-bound processes basically get any service that is left over when all the I/O bound and interactive processes are blocked.

Since Linux does not know a priori whether a task is I/O- or CPU-bound, it relies on continuously maintaining interactivity heuristics. In this manner, Linux distinguishes between static and dynamic priority. The threads' dynamic priority is continuously recalculated, so as to (1) reward interactive threads, and (2) punish CPU-hogging threads. In the O(1) scheduler, the maximum priority bonus is −5, since lower-priority values correspond to higher priority received by the scheduler. The maximum priority penalty is +5. The scheduler maintains a *sleep_avg* variable associated with each task. Whenever a task is awakened, this variable is incremented. Whenever a task is preempted or when its quantum expires, this variable is decremented by the corresponding value. This value is used to dynamically map the task's bonus to values from −5 to +5. The scheduler recalculates the new priority level as a thread is moved from the active to the expired list.

The O(1) scheduling algorithm refers to the scheduler made popular in the early versions of the 2.6 kernel, and was first introduced in the unstable 2.5 kernel. Prior algorithms exhibited poor performance in multiprocessor settings and did not scale well with an increased number of tasks. Since the description presented in the above paragraphs indicates that a scheduling decision can be made through access to the appropriate active list, it can be done in constant O(1) time, independent of the number of processes in the system. However, in spite of the desirable property of constant-time operation, the O(1) scheduler had significant shortcomings. Most notably, the heuristics used to determine the interactivity of a task, and therefore its priority level, were complex and imperfect, and resulted in poor performance for interactive tasks.

To address this issue, Ingo Molnar, who also created the O(1) scheduler, proposed a new scheduler called **CFS** (**Completely Fair Scheduler**). CFS was based on ideas originally developed by Con Kolivas for an earlier scheduler, and was first integrated into the 2.6.23 release of the kernel. It is still the default scheduler for the non-real-time tasks.

The main idea behind CFS is to use a *red-black tree* as the runqueue data structure. Tasks are ordered in the tree based on the amount of time they spend running on the CPU, called *vruntime*. CFS accounts for the tasks' running time with nanosecond granularity. As shown in Fig. 10-10(b), each internal node in the tree corresponds to a task. The children to the left correspond to tasks which had less time on the CPU, and therefore will be scheduled sooner, and the children to the right on the node are those that have consumed more CPU time thus far. The leaves in the tree do not play any role in the scheduler.

The scheduling algorithm can be summarized as follows. CFS always schedules the task which has had least amount of time on the CPU, typically the leftmost node in the tree. Periodically, CFS increments the task's *vruntime* value based on the time it has already run, and compares this to the current leftmost node in the tree. If the running task still has smaller *vruntime*, it will continue to run. Otherwise, it will be inserted at the appropriate place in the red-black tree, and the CPU will be given to task corresponding to the new leftmost node.

To account for differences in task priorities and "niceness," CFS changes the effective rate at which a task's virtual time passes when it is running on the CPU. For lower-priority tasks, time passes more quickly, their *vruntime* value will increase more rapidly, and, depending on other tasks in the system, they will lose the CPU and be reinserted in the tree sooner than if they had a higher priority value. In this manner, CFS avoids using separate runqueue structures for different priority levels.

In summary, selecting a node to run can be done in constant time, whereas inserting a task in the runqueue is done in O(log(*N*)) time, where *N* is the number of tasks in the system. Given the levels of load in current systems, this continues to be acceptable, but as the compute capacity of the nodes, and the number of tasks they can run, increase, particularly in the server space, it is possible that new scheduling algorithms will be needed in the future.

Besides the basic scheduling algorithm, the Linux scheduler includes special features particularly useful for multiprocessor or multicore platforms. First, the runqueue structure is associated with each CPU in the multiprocessing platform. The scheduler tries to maintain benefits from affinity scheduling, and to schedule tasks on the CPU on which they were previously executing. Second, a set of system calls is available to further specify or modify the affinity requirements of a select thread. Finally, the scheduler performs periodic load balancing across runqueues of different CPUs to ensure that the system load is well balanced, while still meeting certain performance or affinity requirements.

The scheduler considers only runnable tasks, which are placed on the appropriate runqueue. Tasks which are not runnable and are waiting on various I/O operations or other kernel events are placed on another data structure, **waitqueue**. A waitqueue is associated with each event that tasks may wait on. The head of the waitqueue includes a pointer to a linked list of tasks and a spinlock. The spinlock is necessary so as to ensure that the waitqueue can be concurrently manipulated through both the main kernel code and interrupt handlers or other asynchronous invocations.

10.3.5 Synchronization in Linux

In the previous section, we mentioned that Linux uses spinlocks to prevent concurrent modifications to data structures like the waitqueues. In fact, the kernel code contains synchronization variables in numerous locations. We will next briefly summarize the synchronization constructs available in Linux.

Earlier Linux kernels had just one **big kernel lock**. This proved highly inefficient, particularly on multiprocessor platforms, since it prevented processes on different CPUs from executing kernel code concurrently. Hence, many new synchronization points were introduced at much finer granularity.

Linux provides several types of synchronization variables, both used internally in the kernel, and available to user-level applications and libraries. At the lowest

level, Linux provides wrappers around the hardware-supported atomic instructions, via operations such as atomic_set and atomic_read. In addition, since modern hardware reorders memory operations, Linux provides memory barriers. Using operations like rmb and wmb guarantees that all read/write memory operations preceding the barrier call have completed before any subsequent accesses take place.

More commonly used synchronization constructs are the higher-level ones. Threads that do not wish to block (for performance or correctness reasons) use spinlocks and spin read/write locks. The current Linux version implements the so-called "ticket-based" spinlock, which has excellent performance on SMP and multicore systems. Threads that are allowed to or need to block use constructs like mutexes and semaphores. Linux supports nonblocking calls like mutex_trylock and sem_trywait to determine the status of the synchronization variable without blocking. Other types of synchronization variables, like futexes, completions, "read-copy-update" (RCU) locks, etc., are also supported. Finally, synchronization between the kernel and the code executed by interrupt-handling routines can also be achieved by dynamically disabling and enabling the corresponding interrupts.

10.3.6 Booting Linux

Details vary from platform to platform, but in general the following steps represent the boot process. When the computer starts, the BIOS performs Power-On-Self-Test (POST) and initial device discovery and initialization, since the OS' boot process may rely on access to disks, screens, keyboards, and so on. Next, the first sector of the boot disk, the **MBR** (**Master Boot Record**), is read into a fixed memory location and executed. This sector contains a small (512-byte) program that loads a standalone program called **boot** from the boot device, such as a SATA or SCSI disk. The *boot* program first copies itself to a fixed high-memory address to free up low memory for the operating system.

Once moved, *boot* reads the root directory of the boot device. To do this, it must understand the file system and directory format, which is the case with some bootloaders such as **GRUB** (**GRand Unified Bootloader**). Other bootloaders, such as Intel's LILO, do not rely on any specific file system. Instead, they need a block map and low-level addresses, which describe physical sectors, heads, and cylinders, to find the relevant sectors to be loaded.

Then *boot* reads in the operating system kernel and jumps to it. At this point, it has finished its job and the kernel is running.

The kernel start-up code is written in assembly language and is highly machine dependent. Typical work includes setting up the kernel stack, identifying the CPU type, calculating the amount of RAM present, disabling interrupts, enabling the MMU, and finally calling the C-language *main* procedure to start the main part of the operating system.

The C code also has considerable initialization to do, but this is more logical than physical. It begins by allocating a message buffer to help debug problems.

As initialization proceeds, messages are written here about what is happening, so that they can be fished out after a boot failure by a special diagnostic program. Think of this as the operating system's cockpit flight recorder (the black box investigators look for after a plane crash).

Next the kernel data structures are allocated. Most are of fixed size, but a few, such as the page cache and certain page table structures, depend on the amount of RAM available.

At this point, the system begins autoconfiguration. Using configuration files telling what kinds of I/O devices might be present, it begins probing the devices to see which ones actually are present. If a probed device responds to the probe, it is added to a table of attached devices. If it fails to respond, it is assumed to be absent and ignored henceforth. Unlike traditional UNIX versions, Linux device drivers do not need to be statically linked and may be loaded dynamically (as can be done in all versions of MS-DOS and Windows, incidentally).

The arguments for and against dynamically loading drivers are interesting and worth stating explicitly. The main argument for dynamic loading is that a single binary can be shipped to customers with divergent configurations and have it automatically load the drivers it needs, possibly even over a network. The main argument against dynamic loading is security. If you are running a secure site, such as a bank's database or a corporate Web server, you probably want to make it impossible for anyone to insert random code into the kernel. The system administrator may keep the operating system sources and object files on a secure machine, do all system builds there, and ship the kernel binary to other machines over a local area network. If drivers cannot be loaded dynamically, this scenario prevents machine operators and others who know the superuser password from injecting malicious or buggy code into the kernel. Furthermore, at large sites, the hardware configuration is known exactly at the time the system is compiled and linked. Changes are rare so having to relink the system when a new device is added is not an issue.

Once all the hardware has been configured, the next thing to do is to carefully handcraft process 0, set up its stack, and run it. Process 0 continues initialization, doing things like programming the real-time clock, mounting the root file system, and creating *init* (process 1) and the page daemon (process 2).

Init checks its flags to see if it is supposed to come up single user or multiuser. In the former case, it forks off a process that executes the shell and waits for this process to exit. In the latter case, it forks off a process that executes the system initialization shell script, */etc/rc*, which can do file system consistency checks, mount additional file systems, start daemon processes, and so on. Then it reads */etc/ttys*, which lists the terminals and some of their properties. For each enabled terminal, it forks off a copy of itself, which does some housekeeping and then executes a program called *getty*.

Getty sets the line speed and other properties for each line (some of which may be modems, for example), and then displays

```
login:
```

on the terminal's screen and tries to read the user's name from the keyboard. When someone sits down at the terminal and provides a login name, *getty* terminates by executing */bin/login*, the login program. *Login* then asks for a password, encrypts it, and verifies it against the encrypted password stored in the password file, */etc/passwd*. If it is correct, *login* replaces itself with the user's shell, which then waits for the first command. If it is incorrect, *login* just asks for another user name. This mechanism is shown in Fig. 10-11 for a system with three terminals.

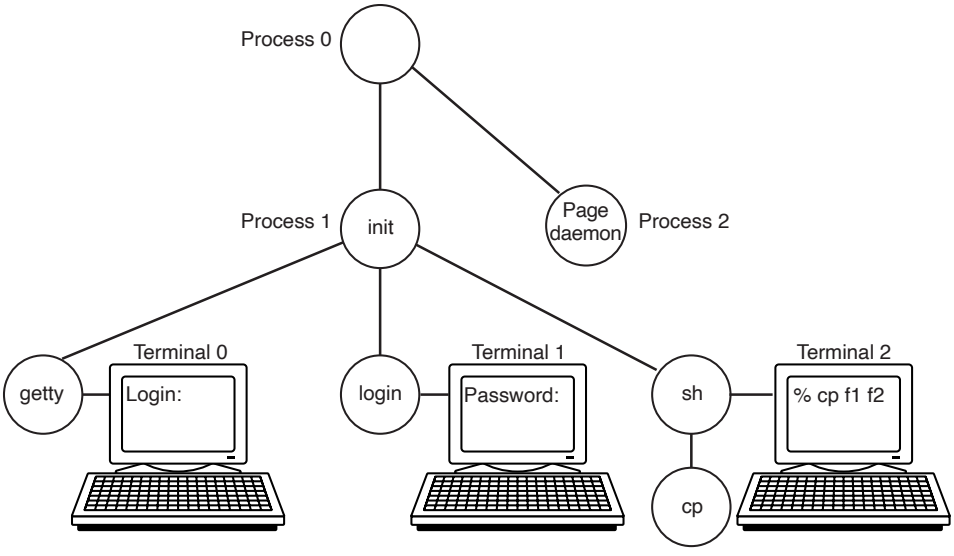

Figure 10-11. The sequence of processes used to boot some Linux systems.

In the figure, the *getty* process running for terminal 0 is still waiting for input. On terminal 1, a user has typed a login name, so *getty* has overwritten itself with *login*, which is asking for the password. A successful login has already occurred on terminal 2, causing the shell to type the prompt (%). The user then typed

```
cp f1 f2
```

which has caused the shell to fork off a child process and have that process execute the *cp* program. The shell is blocked, waiting for the child to terminate, at which time the shell will type another prompt and read from the keyboard. If the user at terminal 2 had typed *cc* instead of *cp*, the main program of the C compiler would have been started, which in turn would have forked off more processes to run the various compiler passes.

10.4 MEMORY MANAGEMENT IN LINUX

The Linux memory model is straightforward, to make programs portable and to make it possible to implement Linux on machines with widely differing memory management units, ranging from essentially nothing (e.g., the original IBM PC) to

sophisticated paging hardware. This is an area of the design that has barely changed in decades. It has worked well so it has not needed much revision. We will now examine the model and how it is implemented.

10.4.1 Fundamental Concepts

Every Linux process has an address space that logically consists of three segments: text, data, and stack. An example process' address space is illustrated in Fig. 10-12(a) as process *A*. The **text segment** contains the machine instructions that form the program's executable code. It is produced by the compiler and assembler by translating the C, C++, or other program into machine code. The text segment is normally read-only. Self-modifying programs went out of style in about 1950 because they were too difficult to understand and debug. Thus the text segment neither grows nor shrinks nor changes in any other way.

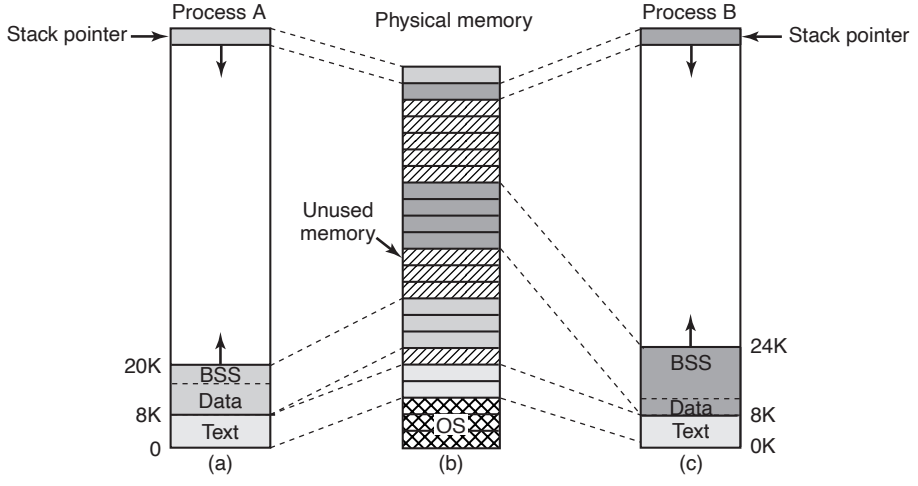

Figure 10-12. (a) Process *A*'s virtual address space. (b) Physical memory. (c) Process *B*'s virtual address space.

The **data segment** contains storage for all the program's variables, strings, arrays, and other data. It has two parts, the initialized data and the uninitialized data. For historical reasons, the latter is known as the **BSS** (historically called **Block Started by Symbol**). The initialized part of the data segment contains variables and compiler constants that need an initial value when the program is started. All the variables in the BSS part are initialized to zero after loading.

For example, in C it is possible to declare a character string and initialize it at the same time. When the program starts up, it expects that the string has its initial value. To implement this construction, the compiler assigns the string a location in the address space, and ensures that when the program is started up, this location contains the proper string. From the operating system's point of view, initialized

data are not all that different from program text—both contain bit patterns produced by the compiler that must be loaded into memory when the program starts.

The existence of uninitialized data is actually just an optimization. When a global variable is not explicitly initialized, the semantics of the C language say that its initial value is 0. In practice, most global variables are not initialized explicitly, and are thus 0. This could be implemented by simply having a section of the executable binary file exactly equal to the number of bytes of data, and initializing all of them, including the ones that have defaulted to 0.

However, to save space in the executable file, this is not done. Instead, the file contains all the explicitly initialized variables following the program text. The uninitialized variables are all gathered together after the initialized ones, so all the compiler has to do is put a word in the header telling how many bytes to allocate. As an example, consider Fig. 10-12(a) again. Here the program text is 8 KB and the initialized data is also 8 KB. The uninitialized data (BSS) is 4 KB. The executable file is only 16 KB (text + initialized data), plus a short header that tells the system to allocate another 4 KB after the initialized data and zero it before starting the program. This trick avoids storing 4 KB of zeros in the executable file.

In order to avoid allocating a physical page frame full of zeros, during initialization Linux allocates a static *zero page*, a write-protected page full of zeros. When a process is loaded, its uninitialized data region is set to point to the zero page. Whenever a process actually attempts to write in this area, the copy-on-write mechanism kicks in, and an actual page frame is allocated to the process.

Unlike the text segment, which cannot change, the data segment can change. Programs modify their variables all the time. Furthermore, many programs need to allocate space dynamically, during execution. Linux handles this by permitting the data segment to grow and shrink as memory is allocated and deallocated. A system call, brk, is available to allow a program to set the size of its data segment. Thus to allocate more memory, a program can increase the size of its data segment. The C library procedure *malloc*, commonly used to allocate memory, makes heavy use of it. The process address-space descriptor contains information on the range of dynamically allocated memory areas in the process, typically called the **heap**.

The third segment is the stack segment. On most machines, it starts at or near the top of the virtual address space and grows down toward 0. For instance, on 32bit x86 platforms, the stack starts at address 0xC0000000, which is the 3-GB virtual address limit visible to the process in user mode. If the stack grows below the bottom of the stack segment, a hardware fault occurs and the operating system lowers the bottom of the stack segment by one page. Programs do not explicitly manage the size of the stack segment.

When a program starts up, its stack is not empty. Instead, it contains all the environment (shell) variables as well as the command line typed to the shell to invoke it. In this way, a program can discover its arguments. For example, when

```
cp src dest
```

is typed, the *cp* program is run with the string "cp src dest" on the stack, so it can find out the names of the source and destination files. The string is represented as an array of pointers to the symbols in the string, to make parsing easier.

When two users are running the same program, such as the editor, it would be possible, but inefficient, to keep two copies of the editor's program text in memory at once. Instead, Linux systems support **shared text segments**. In Fig. 10-12(a) and (c) we see two processes, *A* and *B*, that have the same text segment. In Fig. 10-12(b) we see a possible layout of physical memory, in which both processes share the same piece of text. The mapping is done by the MMU hardware.

Data and stack segments are never shared except after a fork, and then only those pages that are not modified. If either one needs to grow and there is no room adjacent to it to grow into, there is no problem since adjacent virtual pages do not have to map onto adjacent physical pages.

On some computers, the hardware supports separate address spaces for instructions and data. When this feature is available, Linux can use it. For example, on a computer with 32-bit addresses, if this feature is available, there would be 2^{32} bytes of address space for instructions and an additional 2^{32} bytes of address space for the data and stack segments to share. A jump or branch to 0 goes to address 0 of text space, whereas a move from 0 uses address 0 in data space. This feature doubles the address space available.

In addition to dynamically allocating more memory, processes in Linux can access file data through **memory-mapped files**. This feature makes it possible to map a file onto a portion of a process' address space so that the file can be read and written as if it were a byte array in memory. Mapping a file in makes random access to it much easier than using I/O system calls such as read and write. Shared libraries are accessed by mapping them in using this mechanism. In Fig. 10-13, we see a file that is mapped into two processes, at different virtual addresses.

An additional advantage of mapping a file in is that two or more processes can map in the same file at the same time. Writes to the file by any one of them are then instantly visible to the others. In fact, by mapping in a scratch file (which will be discarded after all the processes exit), this mechanism provides a high-bandwidth way for multiple processes to share memory. In the most extreme case, two (or more) processes could map in a file that covers the entire address space, giving a form of sharing that is partway between separate processes and threads. Here the address space is shared (like threads), but each process maintains its own open files and signals, for example, which is not like threads. In practice, however, making two address spaces exactly correspond is never done.

10.4.2 Memory Management System Calls in Linux

POSIX does not specify any system calls for memory management. This topic was considered too ma chine dependent for standardization. Instead, the problem was nicely swept under the rug by saying that programs needing dynamic memory

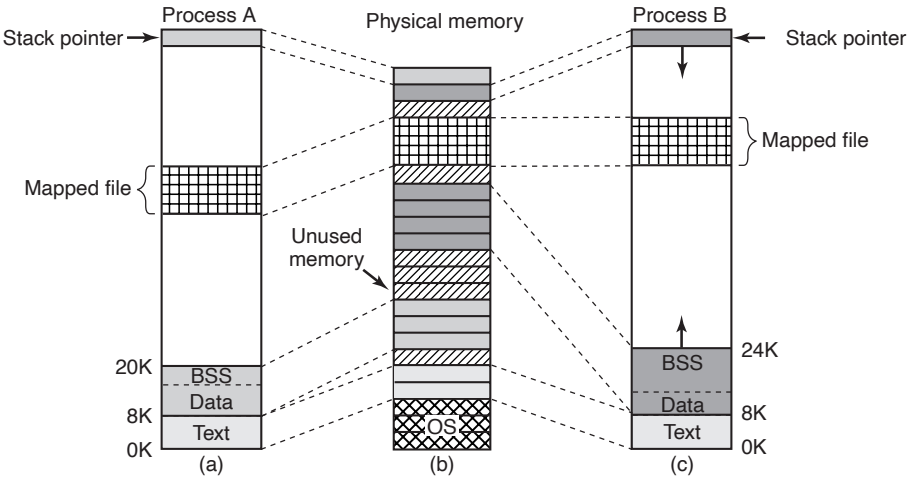

Figure 10-13. Two processes can share a mapped file.

management can use the *malloc* library procedure (defined by the ANSI C standard). How *malloc* is implemented is thus moved outside the scope of the POSIX standard. In some circles, this approach is known as passing the buck.

In practice, most Linux systems have system calls for managing memory. The most common ones are listed in Fig. 10-14. Brk specifies the size of the data segment by giving the address of the first byte beyond it. If the new value is greater than the old one, the data segment becomes larger; otherwise it shrinks.

System call	Description
s = brk(addr)	Change data segment size
a = mmap(addr, len, prot, flags, fd, offset)	Map a file in
s = unmap(addr, len)	Unmap a file

Figure 10-14. Some system calls relating to memory management. The return code *s* is −1 if an error has occurred; *a* and *addr* are memory addresses, *len* is a length, *prot* controls protection, *flags* are miscellaneous bits, *fd* is a file descriptor, and *offset* is a file offset.

The mmap and munmap system calls control memory-mapped files. The first parameter to mmap, *addr*, determines the address at which the file (or portion thereof) is mapped. It must be a multiple of the page size. If this parameter is 0, the system determines the address itself and returns it in *a*. The second parameter, *len*, tells how many bytes to map. It, too, must be a multiple of the page size. The third parameter, *prot*, determines the protection for the mapped file. It can be marked readable, writable, executable, or some combination of these. The fourth parameter, *flags*, controls whether the file is private or sharable, and whether *addr*

is a requirement or merely a hint. The fifth parameter, *fd*, is the file descriptor for the file to be mapped. Only open files can be mapped, so to map a file in, it must first be opened. Finally, *offset* tells where in the file to begin the mapping. It is not necessary to start the mapping at byte 0; any page boundary will do.

The other call, unmap, removes a mapped file. If only a portion of the file is unmapped, the rest remains mapped.

10.4.3 Implementation of Memory Management in Linux

Each Linux process on a 32-bit machine typically gets 3 GB of virtual address space for itself, with the remaining 1 GB reserved for its page tables and other kernel data. The kernel's 1 GB is not visible when running in user mode, but becomes accessible when the process traps into the kernel. The kernel memory typically resides in low physical memory but it is mapped in the top 1 GB of each process virtual address space, between addresses 0xC0000000 and 0xFFFFFFFF (3–4 GB). On most of the current 64-bit x86 machines, only up to 48 bits are used for addressing, implying a theoretical limit of 256 TB for the size of the addressable memory. Linux splits this memory between the kernel and user space, resulting in a maximum 128 TB per-process virtual address space per process. The address space is created when the process is created and is overwritten on an exec system call. Recent hardware enhancement have made it possible to use up to 57 address bits, which further extends the size of the possible addressable memory to 128 PB (Petabytes).

In order to allow multiple processes to share the underlying physical memory, Linux monitors the use of the physical memory, allocates more memory as needed by user processes or kernel components, dynamically maps portions of the physical memory into the address space of different processes, and dynamically brings in and out of memory program executables, files, and other state information as necessary to utilize the platform resources efficiently and to ensure execution progress. The remainder of this section describes the implementation of various mechanisms in the Linux kernel which are responsible for these operations.

Physical Memory Management

Due to idiosyncratic hardware limitations on many systems, not all physical memory can be treated identically, especially with respect to I/O and virtual memory. Linux distinguishes between the following memory zones:

1. **ZONE_DMA** and **ZONE_DMA32**: pages that can be used for DMA.
2. **ZONE_NORMAL**: normal, regularly mapped pages.
3. **ZONE_HIGHMEM**: pages with high-memory addresses, which are not permanently mapped.

The exact boundaries and layout of the memory zones are architecture dependent. On x86 hardware, certain devices can perform DMA operations only in the first 16 MB of address space, hence ZONE_DMA is in the range 0–16 MB. However, on 64-bit machines, there is additional support for those devices that can perform 32-bit DMA operations, and ZONE_DMA32 marks this region. In addition, if the hardware, like the older-generation i386, cannot directly map memory addresses above 896 MB, ZONE_HIGHMEM corresponds to anything above this mark. ZONE_NORMAL is anything in between them. Therefore, on 32-bit x86 platfor ms, the first 896 MB of the Linux address space are directly mapped, whereas the remaining 128 MB of the kernel address space are used to access high memory resgions. On x86_64, ZONE_HIGHMEM is not defined. The kernel maintains a *zone* structure for each of the three zones, and can perform memory allocations for the three zones separately.

Main memory in Linux consists of three parts. The first two parts, the kernel and memory map, are **pinned** in memory (i.e., never paged out). The rest of memory is divided into page frames, each of which can contain a text, data, or stack page, a page-table page, or be on the free list.

The kernel maintains a map of the main memory which contains all information about the use of the physical memory in the system, such as its zones, free page frames, and so forth. The information, illustrated in Fig. 10-15, is organized as follows.

First of all, Linux maintains an array of **page descriptors**, of type *page*, one for each physical page frame in the system, called *mem_map*. Each page descriptor contains a pointer to the address space that it belongs to, in case the page is not free, a pair of pointers which allow it to form doubly linked lists with other descriptors, for instance to keep together all free page frames, and a few other fields. In Fig. 10-15, the page descriptor for page 150 contains a mapping to the address space the page belongs to. Pages 70, 80, and 200 are free, and they are linked together. The size of the page descriptor is 32 bytes, so the mem_map consumes less than 1% of the physical memory (for a page frame of 4 KB).

Since the physical memory is divided into zones, for each zone Linux maintains a *zone descriptor*. The zone descriptor contains information about the memory utilization within each zone, such as the number of active or inactive pages, low and high watermarks to be used by the page-replacement algorithm described later in this chapter, as well as many other fields.

In addition, a zone descriptor contains an array of free areas. The *i*th element in this array identifies the first page descriptor of the first block of 2^i free pages. Since there may be more than one blocks of 2^i free pages, Linux uses the pair of page-descriptor pointers in each page element to link these together. This information is used in the memory-allocation operations. In Fig. 10-15, free_area[0], which identifies all free areas of main memory consisting of only one page frame (since 2^0 is one), points to page 70, the first of the three free areas. The other free blocks of size one can be reached through the links in each of the page descriptors.

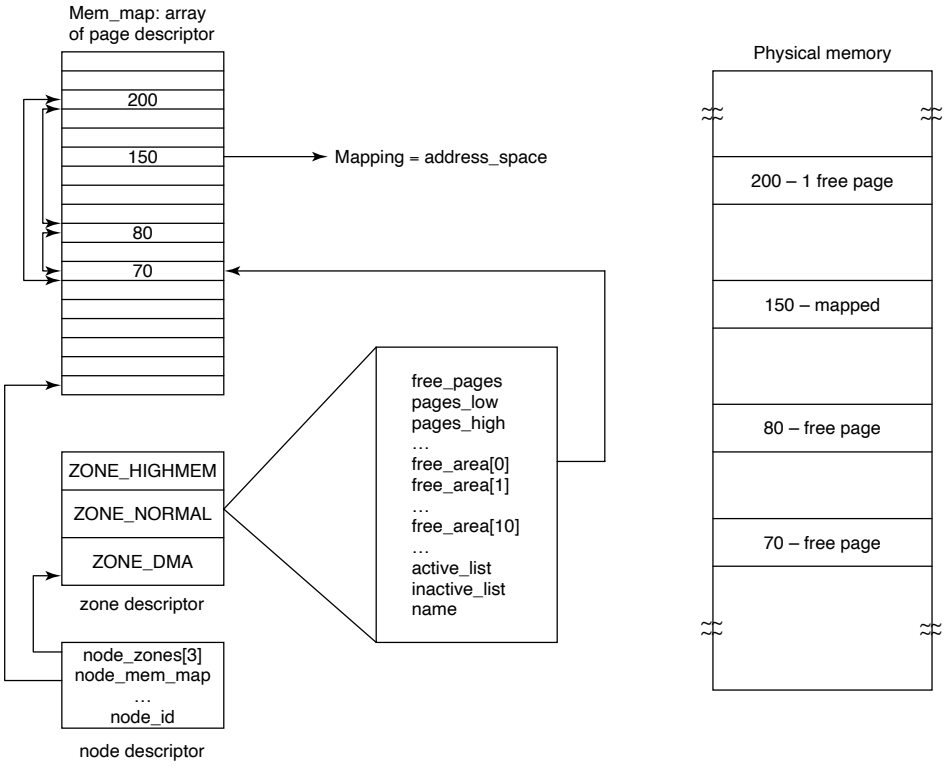

Figure 10-15. Linux' main memory representation.

Finally, since Linux is portable to NUMA architectures (where different memory addresses have different access times), in order to differentiate between physical memory on different nodes (and avoid allocating data structures across nodes), a *node descriptor* is used. Each node descriptor contains information about the memory usage and zones on that particular node. On UMA platforms, Linux describes all memory via one node descriptor. The first few bits within each page descriptor are used to identify the node and the zone that the page frame belongs to.

In order for the paging mechanism to be efficient on both 32- and 64-bit architectures, Linux makes good use of a four-level paging scheme. A three-level paging scheme, originally put into the system for the Alpha, was expanded after Linux 2.6.10, and as of version 2.6.11 a four-level paging scheme is used. Each virtual address is broken up into five fields, as shown in Fig. 10-16. The directory fields are used as an index into the appropriate page directory, of which there is a private one for each process. The value found is a pointer to one of the next-level directories, which are again indexed by a field from the virtual address. The selected

entry in the middle page directory points to the final page table, which is indexed by the page field of the virtual address. The entry found here points to the page needed. On the Pentium, which uses two-level paging, each page's upper and middle directories have only one entry, so the global directory entry effectively chooses the page table to use. Similarly, three-level paging can be used when needed, by setting the size of the upper page directory field to zero. Starting with the 4.14 kernel, five-level page tables are also supported, to leverage the x86-64 hardware extensions originally introduced in the Intel Ice Lake processors.

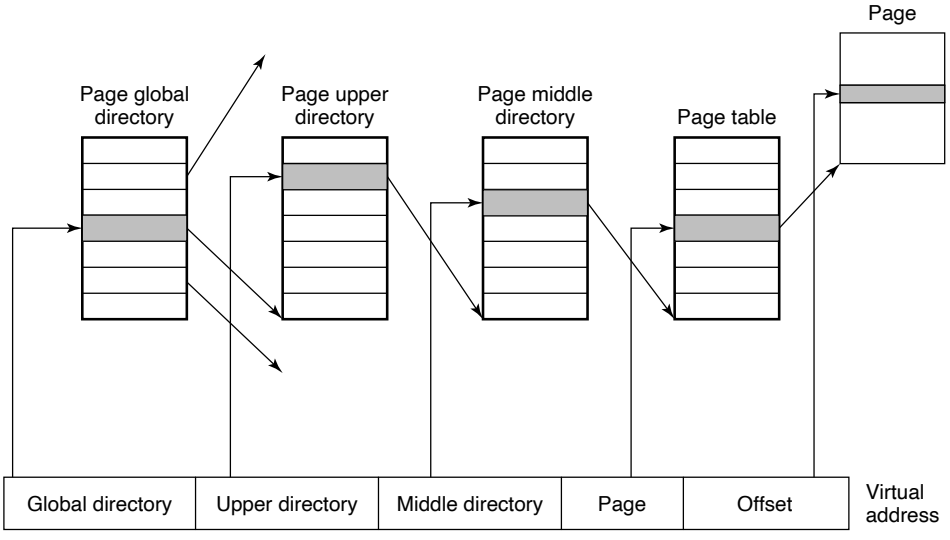

Figure 10-16. Linux uses four-level page tables.

Physical memory is used for various purposes. The kernel itself is fully hardwired; no part of it is ever paged out. The rest of memory is available for user pages, the paging cache, and other purposes. The page cache holds pages containing file blocks that have recently been read or have been read in advance in expectation of being used in the near future, or pages of file blocks which need to be written to disk, such as those which have been created from user-mode processes which have been swapped out to disk. It is dynamic in size and competes for the same pool of pages as the user processes. The paging cache is not really a separate cache, but simply the set of user pages that are no longer needed and are waiting around to be paged out. If a page in the paging cache is reused before it is evicted from memory, it can be reclaimed quickly.

In addition, Linux supports dynamically loaded modules, most often device drivers. These can be of arbitrary size and each one must be allocated a contiguous piece of kernel memory. As a direct consequence of these requirements, Linux

manages physical memory in such a way that it can acquire an arbitrary-sized piece of memory at will. The algorithm it uses is known as the buddy algorithm and is described below.

Memory-Allocation Mechanisms

Linux supports several mechanisms for memory allocation. The main mechanism for allocating new page frames of physical memory is the **page allocator**, which operates using the well-known **buddy algorithm**.

The basic idea for managing a chunk of memory is as follows. Initially memory consists of a single contiguous piece, 64 pages in the simple example of Fig. 10-17(a). When a request for memory comes in, it is first rounded up to a power of 2, say eight pages. The full memory chunk is then divided in half, as shown in (b). Since each of these pieces is still too large, the lower piece is divided in half again (c) and again (d). Now we have a chunk of the correct size, so it is allocated to the caller, as shown shaded in (d).

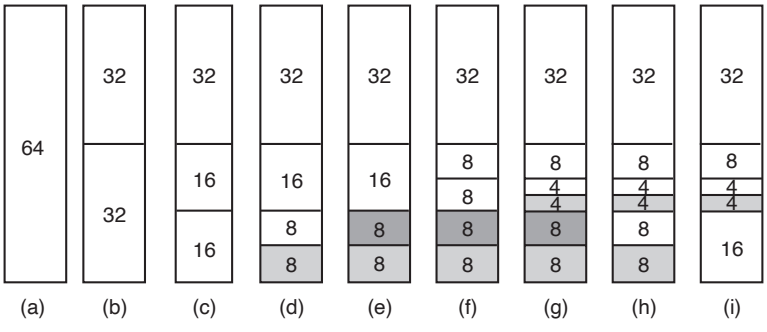

Figure 10-17. Operation of the buddy algorithm.

Now suppose that a second request comes in for eight pages. This can be satisfied directly now (e). At this point, a third request comes in for four pages. The smallest available chunk is split (f) and half of it is claimed (g). Next, the second of the 8-page chunks is released (h). Finally, the other eight-page chunk is released. Since the two adjacent just-freed eight-page chunks came from the same 16-page chunk, they are merged to get the 16-page chunk back (i).

Linux manages memory using the buddy algorithm, with the additional feature of having an array in which the first element is the head of a list of blocks of size 1 unit, the second element is the head of a list of blocks of size 2 units, the next element points to the 4-unit blocks, and so on. In this way, any power-of-2 block can be found quickly.

This algorithm leads to considerable internal fragmentation because if you want a 65-page chunk, you have to ask for and get a 128-page chunk.

To alleviate this problem, Linux has a second memory allocator, the **slab allocator**, which takes chunks using the buddy algorithm but then carves slabs (smaller units) from them and manages the smaller units separately.

Since the kernel frequently creates and destroys objects of certain type (e.g., *task_struct*), it relies on so-called **object caches**. These caches consist of pointers to one or more slab which can store a number of objects of the same type. Each of the slabs may be full, partially full, or empty.

For instance, when the kernel needs to allocate a new process descriptor, that is, a new *task_struct*, it looks in the object cache for task structures, and first tries to find a partially full slab and allocate a new *task_struct* object there. If no such slab is available, it looks through the list of empty slabs. Finally, if necessary, it will allocate a new slab, place the new task structure there, and link this slab with the task-structure object cache. The kmalloc kernel service, which allocates physically contiguous memory regions in the kernel address space, is in fact built on top of the slab and object cache interface described here.

A third memory allocator, vmalloc, is also available and is used when the requested memory needs to be contiguous only in virtual space, not in physical memory. In practice, this is true for most of the requested memory. One exception consists of devices, which live on the other side of the memory bus and the memory management unit, and therefore do not understand virtual addresses. However, the use of vmalloc results in some performance degradation, and it is used primarily for allocating large amounts of contiguous virtual address space, such as for dynamically inserting kernel modules. All these memory allocators are derived from those in System V.

Virtual Address-Space Representation

The virtual address space is divided into homogeneous, contiguous, page-aligned areas or regions. That is to say, each area consists of a run of consecutive pages with the same protection and paging properties. The text segment and mapped files are examples of areas (see Fig. 10-13). There can be holes in the virtual address space between the areas. Any memory reference to a hole results in a fatal page fault. The page size is fixed, for example, 4 KB for the Pentium and 8 KB for the Alpha. Starting with the Pentium, support for page frames of 4 MB was added. On recent 64-bit architectures, Linux can support **huge pages** of 2 MB or 1 GB each. In addition, in a **PAE** (**Physical Address Extension**) mode, which is used on certain 32-bit architectures to increase the process address space beyond 4 GB, page sizes of 2 MB are supported.

Each area is described in the kernel by a *vm_area_struct* entry. All the *vm_area_struct*s for a process are linked together in a list sorted on virtual address so that all the pages can be found. When the list gets too long (more than 32 entries), a tree is created to speed up searching it. The *vm_area_struct* entry lists the area's properties. These properties include the protection mode (e.g., read only

or read/write), whether it is pinned in memory (not pageable), and which direction it grows in (up for data segments, down for stacks).

The *vm_area_struct* also records whether the area is private to the process or shared with one or more other processes. After a fork, Linux makes a copy of the area list for the child process, but sets up the parent and child to point to the same page tables. The areas are marked as read/write, but the pages themselves are marked as read only. If either process tries to write on a page, a protection fault occurs and the kernel sees that the area is logically writable but the page is not writeable, so it gives the process a copy of the page and marks it read/write. This mechanism is how copy on write is implemented.

The *vm_area_struct* also records whether the area has backing storage on disk assigned, and if so, where. Text segments use the executable binary as backing storage and memory-mapped files use the disk file as backing storage. Other areas, such as the stack, do not have backing storage assigned until they have to be paged out.

A top-level memory descriptor, *mm_struct*, gathers information about all virtual-memory areas belonging to an address space, information about the different segments (text, data, stack), about users sharing this address space, and so on. All *vm_area_struct* elements of an address space can be accessed through their memory descriptor in two ways. First, they are organized in linked lists ordered by virtual-memory addresses. This way is useful when all virtual-memory areas need to be accessed, or when the kernel is searching to allocate a virtual-memory region of a specific size. In addition, the *vm_area_struct* entries are organized in a binary "red-black" tree, a data structure optimized for fast lookups. This method is used when a specific virtual memory needs to be accessed. By enabling access to elements of the process address space via these two methods, Linux uses more state per process, but allows different kernel operations to use the access method which is more efficient for the task at hand.

10.4.4 Paging in Linux

Early UNIX systems relied on a **swapper process** to move entire processes between memory and disk whenever not all active processes could fit in the physical memory. Linux, like other modern UNIX versions, no longer moves entire processes. The main memory management unit is a page, and almost all memory-management components operate on a page granularity. The swapping subsystem also operates on page granularity and is tightly coupled with the **page frame reclaiming algorithm**, described later in this section.

The basic idea behind paging in Linux is simple: a process need not be entirely in memory in order to run. All that is actually required is the user structure and the page tables. If these are swapped in, the process is deemed "in memory" and can be scheduled to run. The pages of the text, data, and stack segments are brought in

dynamically, one at a time, as they are referenced. If the user structure and page table are not in memory, the process cannot be run until the swapper brings them in.

Paging is implemented partly by the kernel and partly by a new process called the **page daemon**. The page daemon is process 2 (process 0 is the idle process—traditionally called the swapper—and process 1 is *init*, as shown in Fig. 10-11). Like all daemons, the page daemon runs periodically. Once awake, it looks around to see if there is any work to do. If it sees that the number of pages on the list of free memory pages is too low, it starts freeing up more pages.

Linux is a fully demand-paged system with no prepaging and no working-set concept (although there is a call in which a user can give a hint that a certain page may be needed soon, in the hope it will be there when needed). Text segments and mapped files are paged to their respective files on disk. Everything else is paged to either the paging partition (if present) or one of the fixed-length paging files, called the **swap area**. Paging files can be added and removed dynamically and each one has a priority. Paging to a separate partition, accessed as a raw device, is more efficient than paging to a file for several reasons. First, the mapping between file blocks and disk blocks is not needed (saves disk I/O reading indirect blocks). Second, the physical writes can be of any size, not just the file block size. Third, a page is always written contiguously to disk; with a paging file, it may or may not be.

Pages are not allocated on the paging device or partition until they are needed. Each device and file starts with a bitmap telling which pages are free. When a page without backing store has to be tossed out of memory, the highest-priority paging partition or file that still has space is chosen and a page allocated on it. Normally, the paging partition, if present, has higher priority than any paging file. The page table is updated to reflect that the page is no longer present in memory (e.g., the page-not-present bit is set) and the disk location is written into the page-table entry.

The Page Replacement Algorithm

Page replacement works as follows. Linux tries to keep some pages free so that they can be claimed as needed. Of course, this pool must be continually replenished. The **PFRA** (**Page Frame Reclaiming Algorithm**) algorithm is how this happens.

First of all, Linux distinguishes between four different types of pages: *unreclaimable*, *swappable*, *syncable*, and *discardable*. Unreclaimable pages, which include reserved or locked pages, kernel mode stacks, and the like, may not be paged out. Swappable pages must be written back to the swap area or the paging disk partition before the page can be reclaimed. Syncable pages must be written back to disk if they have been marked as dirty. Finally, discardable pages can be reclaimed immediately.

At boot time, *init* starts up a page daemon, *kswapd*, for each memory node, and configures them to run periodically. Each time *kswapd* awakens, it checks to see if there are enough free pages available, by comparing the low and high watermarks with the current memory usage for each memory zone. If there is enough memory, it goes back to sleep, although it can be awakened early if more pages are suddenly needed. If the available memory for any of the zones ever falls below a threshold, *kswapd* initiates the page frame reclaiming algorithm. During each run, only a certain target number of pages is reclaimed, typically a maximum of 32. This number is limited to control the I/O pressure (the number of disk writes created during the PFRA operations). Both the number of reclaimed pages and the total number of scanned pages are configurable parameters.

Each time PFRA executes, it first tries to reclaim easy pages, then proceeds with the more difficult ones. Many people also grab the low-hanging fruit first. Discardable and unreferenced pages can be reclaimed immediately by moving them onto the zone's freelist. Next it looks for pages with backing store which have not been referenced recently, using a clock-like algorithm. Following are shared pages that none of the users seems to be using much. The challenge with shared pages is that, if a page entry is reclaimed, the page tables of all address spaces originally sharing that page must be updated in a synchronous manner. Linux maintains efficient tree-like data structures to easily find all users of a shared page. Ordinary user pages are searched next, and if chosen to be evicted, they must be scheduled for write in the swap area. The **swappiness** of the system, that is, the ratio of pages with backing store vs. pages which need to be swapped out selected during PFRA, is a tunable parameter of the algorithm. Finally, if a page is invalid, absent from memory, shared, locked in memory, or being used for DMA, it is skipped.

PFRA uses a clock-like algorithm to select old pages for eviction within a certain category. At the core of this algorithm is a loop which scans through each zone's active and inactive lists, trying to reclaim different kinds of pages, with different urgencies. The urgency value is passed as a parameter telling the procedure how much effort to expend to reclaim some pages. Usually, this means how many pages to inspect before giving up.

During PFRA, pages are moved between the active and inactive list in the manner described in Fig. 10-18. To maintain some heuristics and try to find pages which have not been referenced and are unlikely to be needed in the near future, PFRA maintains two flags per page: active/inactive, and referenced or not. These two flags encode four states, as shown in Fig. 10-18. During the first scan of a set of pages, PFRA first clears their reference bits. If during the second run over the page it is determined that it has been referenced, it is advanced to another state, from which it is less likely to be reclaimed. Otherwise, the page is moved to a state from where it is more likely to be evicted.

Pages on the inactive list, which have not been referenced since the last time they were inspected, are the best candidates for eviction. They are pages with both

PG_active and *PG_referenced* set to zero in Fig. 10-18. However, if necessary, pages may be reclaimed even if they are in some of the other states. The *refill* arrows in Fig. 10-18 illustrate this fact.

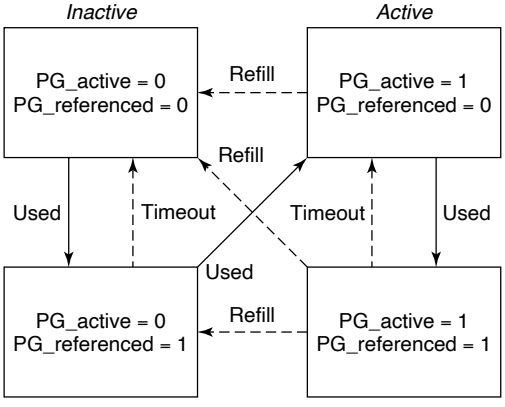

Figure 10-18. Page states considered in the page-frame replacement algorithm.

The reason PRFA maintains pages in the inactive list although they might have been referenced is to prevent situations such as the following. Consider a process which makes periodic accesses to different pages, with a 1-hour period. A page accessed since the last loop will have its reference flag set. However, since it will not be needed again for the next hour, there is no reason not to consider it as a candidate for reclamation.

The actual step of reclaiming memory pages is performed by *kernel worker threads*. These threads either (1) wake up periodically, typically every 500 msec, to write back to disk very old dirty pages, or (2) are explicitly awakened by the kernel when available memory levels fall below a certain threshold, to write back dirty pages from the page cache to disk. Dirty pages may also be written out to disk on explicit requests for synchronization, via systems calls such as sync, fsync, or fdatasync. Older Linux versions used two separate daemons: *kupdate*, for old-page write back, and *bdflush*, for page write back under low memory conditions. In the 2.4 kernel, this functionality was integrated in the *pdflush* threads. The choice of multiple threads was made in order to hide long disk latencies. Later, the *pdflush* threads were replaced first by per-block device *flusher* threads, until the writeback (and other) functionality was all assigned to the kernel worker threads.

10.5 INPUT/OUTPUT IN LINUX

The I/O system in Linux is fairly straightforward and the same as in other UNICES. Basically, all I/O devices are made to look like files and are accessed as such with the same read and write system calls that are used to access all ordinary

files. In some cases, device parameters must be set, and this is done using a special system call. We will study these issues in the following sections.

10.5.1 Fundamental Concepts

Like all computers, those running Linux have I/O devices such as disks, printers, and networks connected to them. Some way is needed to allow programs to access these devices. Although various solutions are possible, the Linux one is to integrate the devices into the file system as what are called **special files**. Each I/O device is assigned a path name, usually in */dev*. For example, a disk might be */dev/hd1*, a printer might be */dev/lp*, and the network might be */dev/net*.

These special files can be accessed the same way as any other files. No special commands or system calls are needed. The usual open, read, and write system calls will do just fine. For example, the command

```
cp file /dev/lp
```

copies the *file* to printer, causing it to be printed (assuming that the user has permission to access */dev/lp*). Programs can open, read, and write special files exactly the same way as they do regular files. In fact, *cp* in the above example is not even aware that it is printing. In this way, no special mechanism is needed for doing I/O.

Special files are divided into two categories, block and character. A **block special file** is one consisting of a sequence of numbered blocks. The key property of the block special file is that each block can be individually addressed and accessed. In other words, a program can open a block special file and read, say, block 124 without first having to read blocks 0 to 123. Block special files are typically used for disks (and SSDs, of course).

Character special files are normally used for devices that input or output a character stream. Keyboards, printers, networks, mice, plotters, and most other I/O devices that accept or produce data for people use character special files. It is not possible (or even meaningful) to seek to block 124 on a mouse.

Associated with each special file is a device driver that handles the corresponding device. Each driver has what is called a **major device** number that serves to identify it. If a driver supports multiple devices, say, two disks of the same type, each disk has a **minor device** number that identifies it. Together, the major and minor device numbers uniquely specify every I/O device. In few cases, a single driver handles two closely related devices. For example, the driver corresponding to */dev/tty* controls both the keyboard and the screen, often thought of as a single device, the terminal.

Although most character special files cannot be randomly accessed, they often need to be controlled in ways that block special files do not. Consider, for example, input typed on the keyboard and displayed on the screen. When a user makes a typing error and wants to erase the last character typed, he presses some key. Some

people prefer to use backspace, and others prefer DEL. Similarly, to erase the entire line just typed, many conventions abound. Traditionally @ was used, but with the spread of email (which uses @ within email address), many systems have adopted CTRL-U or some other character. Likewise, to interrupt the running program, some special key must be hit. Here, too, different people have different preferences. CTRL-C is a common choice, but it is not universal.

Rather than making a choice and forcing everyone to use it, Linux allows all these special functions and many others to be customized by the user. A special system call is generally provided for setting these options. This system call also handles tab expansion, enabling and disabling of character echoing, conversion between carriage return and line feed, and similar items. The system call is not permitted on regular files or block special files.

10.5.2 Networking

Another example of I/O is networking, as pioneered by Berkeley UNIX and taken over by Linux more or less verbatim. The key concept in the Berkeley design is the **socket**. Sockets are analogous to mailboxes and telephone wall sockets in that they allow users to interface to the network, just as mailboxes allow people to interface to the postal system and telephone wall sockets allow them to plug in telephones and connect to the telephone system. The sockets' position is shown in Fig. 10-19.

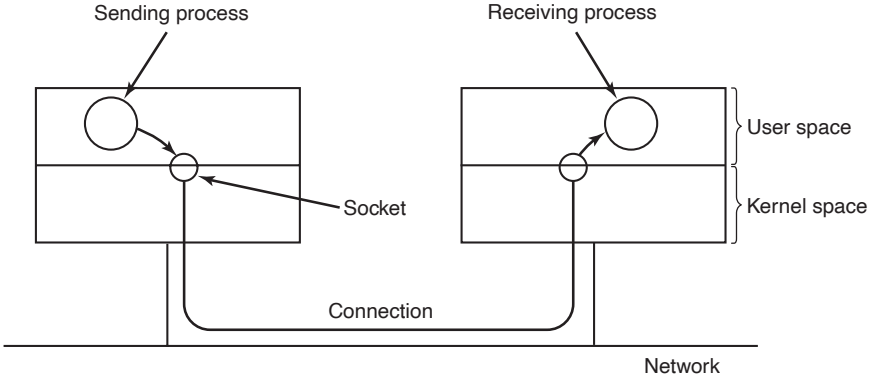

Figure 10-19. The uses of sockets for networking.

Sockets can be created and destroyed dynamically. Creating a socket returns a file descriptor, which is needed for establishing a connection, reading data, writing data, and releasing the connection.

Each socket supports a particular type of networking, specified when the socket is created. The most common types are as follows:

1. Reliable connection-oriented byte stream.
2. Reliable connection-oriented packet stream.
3. Unreliable packet transmission.

The first socket type allows two processes on different machines to establish the equivalent of a pipe between them. Bytes are pumped in at one end and they come out in the same order at the other. The system guarantees that all bytes that are sent correctly arrive and in the same order they were sent.

The second type is rather similar to the first one, except that it preserves packet boundaries. If the sender makes five separate calls to write, each for 512 bytes, and the receiver asks for 2560 bytes, with a type 1 socket all 2560 bytes will be returned at once. With a type 2 socket, only 512 bytes will be returned. Four more calls are needed to get the rest. The third type of socket is used to give the user access to the raw network. This type is especially useful for real-time applications, and for those situations in which the user wants to implement a specialized error handling scheme. Packets may be lost or reordered by the network. There are no guarantees, as in the first two cases. The advantage of this mode is higher performance, which sometimes outweighs reliability (e.g., for multimedia delivery, in which being fast counts for more than being right).

When a socket is created, one of the parameters specifies the protocol to be used for it. For reliable byte streams, the most popular protocol is **TCP** (**Transmission Control Protocol**). For unreliable packet-oriented transmission, **UDP** (**User Datagram Protocol**) is the usual choice. Both of these are layered on top of **IP** (**Internet Protocol**). All of these protocols originated with the U.S. Dept. of Defense's ARPANET, and now form the basis of the Internet. There is no common protocol for reliable packet streams.

Before a socket can be used for networking, it must have an address bound to it. This address can be in one of several naming domains. The most common one is the Internet naming domain, which uses 32-bit integers for naming endpoints in Version 4 and 128-bit integers in Version 6 (Version 5 was an experimental system that never made it to the major leagues).

Once sockets have been created on both the source and destination computers, a connection can be established between them (for connection-oriented communication). One party makes a listen system call on a local socket, which creates a buffer and blocks until data arrive. The other makes a connect system call, giving as parameters the file descriptor for a local socket and the address of a remote socket. If the remote party accepts the call, it creates a new socket (since it may need the original one to continue to listen for other connection requests), and the system then establishes a connection between the caller's socket and the newly created remote socket.

Once a connection has been established, it functions analogously to a pipe. A process can read and write from it using the file descriptor for its local socket.

When the connection is no longer needed, it can be closed in the usual way, via the close system call.

10.5.3 Input/Output System Calls in Linux

Each I/O device in a Linux system generally has a special file associated with it. Most I/O can be done by just using the proper file, eliminating the need for special system calls. Nevertheless, sometimes there is a need for something that is device specific. Prior to POSIX, most UNIX systems had a system call ioctl that performed a large number of device-specific actions on special files. Over the course of the years, it had gotten to be quite a mess. POSIX cleaned it up by splitting its functions into separate function calls primarily for terminal devices. In Linux and modern UNIX systems, whether each one is a separate system call or they share a single system call or something else is implementation dependent.

The first four calls listed in Fig. 10-20 are used to set and get the terminal speed. Different calls are provided for input and output because some modems operate at split speed. For example, old videotex systems allowed people to access public databases with short requests from the home to the server at 75 bits/sec with replies coming back at 1200 bits/sec. This standard was adopted at a time when 1200 bits/sec both ways was too expensive for home use. Times change in the networking world. This asymmetry still persists, with some telephone companies offering inbound service at 40 Mbps and outbound service at 10 Mbps, or some other asymmetric arrangement. With fiber optics, the inbound and outbound speeds are generally the same, for example, 500/500.

Function call	Description
s = cfsetospeed(&termios, speed)	Set the output speed
s = cfsetispeed(&termios, speed)	Set the input speed
s = cfgetospeed(&termios, speed)	Get the output speed
s = cfgtetispeed(&termios, speed)	Get the input speed
s = tcsetattr(fd, opt, &termios)	Set the attributes
s = tcgetattr(fd, &termios)	Get the attributes

Figure 10-20. The main POSIX calls for managing the terminal.

The last two calls in the list are for setting and reading back all the special characters used for erasing characters and lines, interrupting processes, and so on. In addition, they enable and disable echoing, handle flow control, and perform similar functions. Additional I/O function calls also exist, but they are somewhat specialized, so we will not discuss them further. In addition, ioctl is still available.

10.5.4 Implementation of Input/Output in Linux

I/O in Linux is implemented by a collection of device drivers, one per device type. The function of the drivers is to isolate the rest of the system from the idiosyncrasies of the hardware. By providing standard interfaces between the drivers and the rest of the operating system, most of the I/O system can be put into the machine-independent part of the kernel.

When the user accesses a special file, the file system determines the major and minor device numbers belonging to it and whether it is a block special file or a character special file. The major device number is used to index into one of two internal hash tables containing data structures for character or block devices. The structure thus located contains pointers to the procedures to call to open the device, read the device, write the device, and so on. The minor device number is passed as a parameter. Adding a new device type to Linux means adding a new entry to one of these tables and supplying the corresponding procedures to handle the various operations on the device.

Some of the operations which may be associated with different character devices are shown in Fig. 10-21. Each row refers to a single I/O device (i.e., a single driver). The columns represent the functions that all character drivers must support. Several other functions also exist. When an operation is performed on a character special file, the system indexes into the hash table of character devices to select the proper structure, then calls the corresponding function to have the work performed. Thus each of the file operations contains a pointer to a function contained in the corresponding driver.

Device	Open	Close	Read	Write	Ioctl	Other
Null	null	null	null	null	null	...
Memory	null	null	mem_read	mem_write	null	...
Keyboard	k_open	k_close	k_read	error	k_ioctl	...
Tty	tty_open	tty_close	tty_read	tty_write	tty_ioctl	...
Printer	lp_open	lp_close	error	lp_write	lp_ioctl	...

Figure 10-21. Some of the file operations supported for typical character devices.

Each driver is split into two parts, both of which are part of the Linux kernel and both of which run in kernel mode. The top half runs in the context of the caller and interfaces to the rest of Linux. The bottom half runs in kernel context and interacts with the device. Drivers are allowed to make calls to kernel procedures for memory allocation, timer management, DMA control, and other things. The set of kernel functions that may be called is defined in a document called the **Driver-Kernel Interface**. Writing device drivers for Linux is covered in detail in Cooperstein (2009) and Corbet et al. (2009).

The I/O system is split into two major components: the handling of block special files and the handling of character special files. We will now look at each of these components in turn.

The goal of the part of the system that does I/O on block special files (e.g., disks) is to minimize the number of transfers that must be done. To accomplish this goal, Linux has a **cache** between the disk drivers and the file system, as illustrated in Fig. 10-22. Prior to the 2.2 kernel, Linux maintained completely separate page and buffer caches, so a file residing in a disk block could be cached in both caches. Newer versions of Linux have a unified cache. A *generic block layer* holds these components together, performs the necessary translations between disk sectors, blocks, buffers and pages of data, and enables the operations on them.

The cache is a table in the kernel for holding thousands of the most recently used blocks. When a block is needed from a disk for whatever reason (i-node, directory, or data), a check is first made to see if it is in the cache. If it is present in the cache, the block is taken from there and a disk access is avoided, thereby resulting in great improvements in system performance.

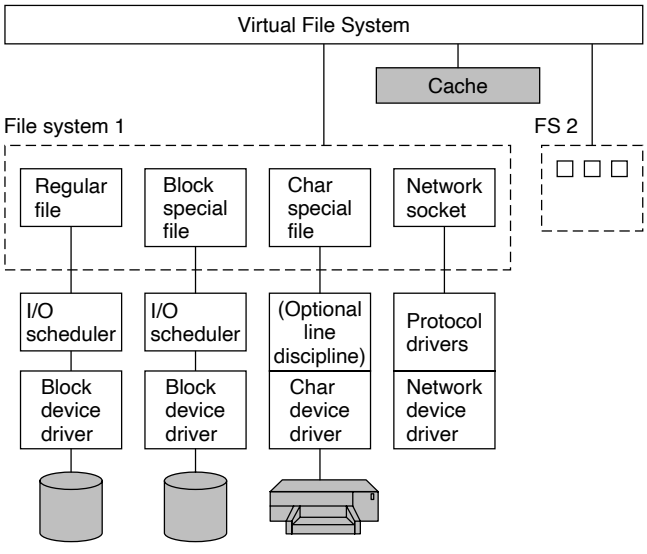

Figure 10-22. The Linux I/O system showing one file system in detail.

If the block is not in the page cache, it is read from the disk into the cache and from there copied to where it is needed. Since the page cache has room for only a fixed number of blocks, the page-replacement algorithm described in the previous section is invoked.

The page cache works for writes as well as for reads. When a program writes a block, it goes to the cache, not to the disk. The kernel worker threads will flush the block to disk in the event the cache grows above a specified value. In addition, to

avoid having blocks stay too long in the cache before being written to the disk, all dirty blocks are written to the disk every 30 seconds.

New types of storage devices are memory-like, in that they can be accessed more quickly and at smaller block granularity (even few bytes or a cacheline). In such cases, moving data in-and-out between the storage device and an in-memory cache is an overkill. Starting with the 4.0 kernel, Linux supports **DAX** (**Direct Access for files**). With DAX, the cache is removed and reads and writes are directly issued to the storage device.

In order to reduce the latency of repetitive disk-head movements, or of random I/O accesses in general, Linux relies on an **I/O scheduler**. Its purpose is to reorder or bundle read/write requests to block devices. There are many scheduler variants, optimized for different types of workloads. The basic Linux scheduler is based on the original **Linux elevator scheduler**. The operations of the elevator scheduler can be summarized as follows: Disk operations are sorted in a doubly linked list, ordered by the address of the sector of the disk request. New requests are inserted in this list in a sorted manner. This prevents repeated costly disk-head movements. The request list is subsequently *merged* so that adjacent operations are issued via a single disk request. The basic elevator scheduler can lead to starvation. Therefore, the revised version of the Linux disk scheduler includes two additional lists, maintaining read or write operations ordered by their deadlines. The default deadlines are 0.5 sec for reads and 5 sec for writes. If a system-defined deadline for the oldest write operation is about to expire, that write request will be serviced before any of the requests on the main doubly linked list.

In addition to regular disk files, there are also block special files, sometimes called **raw block files**. These files allow programs to access the disk using absolute block numbers, without regard to the file system. They are most often used for things like paging and system maintenance.

The interaction with character devices is simple. Since character devices produce or consume streams of characters, or bytes of data, support for random access makes little sense. One exception is the use of **line disciplines**. A line discipline can be associated with a terminal device, represented via the structure *tty_struct*, and it represents an interpreter for the data exchanged with the terminal device. For instance, local line editing can be done (i.e., erased characters and lines can be removed), carriage returns can be mapped onto line feeds, and other special processing can be completed. However, if a process wants to interact on every character, it can put the line in raw mode, in which case the line discipline will be bypassed. Not all devices have line disciplines.

Output works in a similar way, expanding tabs to spaces, converting line feeds to carriage returns + line feeds, adding filler characters following carriage returns on slow mechanical terminals, and so on. Like input, output can go through the line discipline (cooked mode) or bypass it (raw mode). Raw mode is especially useful when sending binary data to other computers over a serial line and for GUIs. Here, no conversions are desired.

The interaction with **network devices** is different. While network devices also produce/consume streams of characters, their asynchronous nature makes them less suitable for easy integration under the same interface as other character devices. The networking device driver produces packets consisting of multiple bytes of data, along with network headers. These packets are then routed through a series of network protocol drivers, and ultimately are passed to the user-space application. A key data structure is the socket buffer structure, *skbuff*, which is used to represent portions of memory filled with packet data. The data in an *skbuff* buffer do not always start at the start of the buffer. As they are being processed by various protocols in the networking stack, protocol headers may be removed, or added. The user processes interact with networking devices via sockets, which in Linux support the original BSD socket API. The protocol drivers can be bypassed and direct access to the underlying network device is enabled via *raw_sockets*. Only the superuser is allowed to create raw sockets.

10.5.5 Modules in Linux

For decades, UNIX device drivers were statically linked into the kernel so they were all present in memory whenever the system was booted. Given the environment in which UNIX grew up, commonly departmental minicomputers and then high-end workstations, with their small and unchanging sets of I/O devices, this scheme worked well. Basically, a computer center built a kernel containing drivers for the I/O devices it actually had and that was it. If next year the center bought a new disk, it relinked the kernel. No big deal.

With the arrival of Linux on the PC platform, suddenly all that changed. The number of I/O devices available on the PC is orders of magnitude larger than on any minicomputer. In addition, although all Linux users have (or can easily get) the full source code, probably the vast majority would have considerable difficulty adding a driver, updating all the device-driver related data structures, relinking the kernel, and then installing it as the bootable system (not to mention dealing with the aftermath of building a kernel that does not boot).

Linux solved this problem with the concept of **loadable modules**. These are chunks of code that can be loaded into the kernel while the system is running. Most commonly these are character or block device drivers, but they can also be entire file systems, network protocols, performance monitoring tools, or anything else desired.

When a module is loaded, several things have to happen. First, the module has to be relocated on the fly, during loading. Second, the system has to check to see if the resources the driver needs are available (e.g., interrupt request levels) and if so, mark them as in use. Third, any interrupt vectors that are needed must be set up. Fourth, the appropriate driver switch table has to be updated to handle the new major device type. Finally, the driver is allowed to run to perform any device-specific initialization it may need. Once all these steps are completed, the driver is

fully installed, the same as any statically installed driver. Other modern UNIX systems now also support loadable modules.

It is worth noting that loadable modules are a security nightmare. Putting a piece of foreign code that may or may not have been vetted carefully and which might contain security holes and backdoors into the kernel can create huge security problems. Loadable modules should only be fetched from a source known to be completely trustworthy.

10.6 THE LINUX FILE SYSTEM

The most visible part of any operating system, including Linux, is the file system. In the following sections, we will examine the basic ideas behind the Linux file system, the system calls, and how the file system is implemented. Some of these ideas derive from MULTICS, and many of them have been copied by MS-DOS, Windows, and other systems, but others are unique to UNIX-based systems. The Linux design is especially interesting because it clearly illustrates the principle of *Small is Beautiful*. With minimal mechanism and a very limited number of system calls, Linux nevertheless provides a powerful and elegant file system.

10.6.1 Fundamental Concepts

The initial Linux file system was the MINIX 1 file system. However, because it limited file names to 14 characters (in order to be compatible with UNIX Version 7) and its maximum file size was 64 MB (which was overkill on the 10-MB hard disks of its era), there was interest in better file systems almost from the beginning of the Linux development, which began about 5 years after MINIX 1 was released. The first improvement was the ext file system, which allowed file names of 255 characters and files of 2 GB, but it was slower than the MINIX 1 file system, so the search continued for a while. Eventually, the ext2 file system was invented, with long file names, long files, and better performance, and it has become the main file system. However, Linux supports several dozen file systems using the Virtual File System (VFS) layer (described in the next section). When Linux is linked, a choice is offered of which file systems should be built into the kernel. Others can be dynamically loaded as modules during execution, if need be.

A Linux file is a sequence of 0 or more bytes containing arbitrary information. No distinction is made between ASCII files, binary files, or any other kinds of files. The meaning of the bits in a file is entirely up to the file's owner. The system does not care. File names are limited to 255 characters, and all the ASCII characters except NUL are allowed in file names, so a file name consisting of three carriage returns is a legal file name (but not an especially convenient one).

By convention, many programs (for example, compilers) expect file names to consist of a base name and an extension, separated by a dot (which counts as a

character). Thus *prog.c* is typically a C program, *prog.py* is typically a Python program, and *prog.o* is usually an object file (compiler output). These conventions are not enforced by the operating system but some compilers and other programs expect them. Extensions may be of any length, and files may have multiple extensions, as in *prog.java.gz*, which is probably a *gzip* compressed Java program.

Files can be grouped together in directories for convenience. Directories are stored as files and to a large extent can be treated like files. Directories can contain subdirectories, leading to a hierarchical file system. The root directory is called / and always contains several subdirectories. The / character is also used to separate directory names, so that the name */usr/ast/x* denotes the file *x* located in the directory *ast*, which itself is in the */usr* directory. Some of the major directories near the top of the tree are shown in Fig. 10-23.

Directory	Contents
bin	Binary (executable) programs
dev	Special files for I/O devices
etc	Miscellaneous system files
lib	Libraries
usr	User directories

Figure 10-23. Some important directories found in most Linux systems.

There are two ways to specify file names in Linux, both to the shell and when opening a file from inside a program. The first way is by means of an **absolute path**, which means telling how to get to the file starting at the root directory. An example of an absolute path is */usr/ast/books/mos5/chap-10*. This tells the system to look in the root directory for a directory called *usr*, then look there for another directory, *ast*. In turn, this directory contains a directory *books*, which contains the directory *mos5*, which contains the file *chap-10*.

Absolute path names are often long and inconvenient. For this reason, Linux allows users and processes to designate the directory in which they are currently working as the **working directory**. Path names can also be specified relative to the working directory. A path name specified relative to the working directory is a **relative path**. For example, if */usr/ast/books/mos5* is the working directory, then the shell command

```
cp chap-10 backup-10
```

has exactly the same effect as the longer command

```
cp  /usr/ast/books/mos5/chap-10  /usr/ast/books/mos5/backup-10
```

It frequently occurs that a user needs to refer to a file that belongs to another user, or at least is located elsewhere in the file tree. For example, if two users are sharing a file, it will be located in a directory belonging to one of them, so the

other will have to use an absolute path name to refer to it (or change the working directory). If this is long enough, it may become irritating to have to keep typing it. Linux provides a solution by allowing users to make a new directory entry that points to an existing file. Such an entry is called a **link**.

As an example, consider the situation of Fig. 10-24(a). Aron and Nathan are working together on a project, and each of them needs access to the other's files. If Aron has */usr/aron* as his working directory, he can refer to the file *x* in Nathan's directory as */usr/nathan/x*. Alternatively, Aron can create a new entry in his directory, as shown in Fig. 10-24(b), after which he can use *x* to mean */usr/nathan/x*.

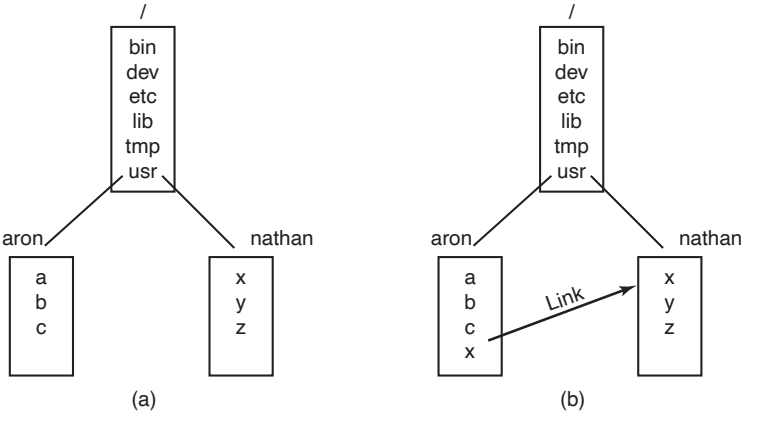

Figure 10-24. (a) Before linking. (b) After linking.

In the example just discussed, we suggested that before linking, the only way for Aron to refer to Nathan's file *x* was by using its absolute path. Actually, this is not really true. When a directory is created, two entries, . and .., are automatically made in it. The former refers to the working directory itself. The latter refers to the directory's parent, that is, the directory in which it itself is listed. Thus from */usr/aron*, another path to Nathan's file *x* is *../nathan/x* .

In addition to regular files, Linux also supports character special files and block special files. Character special files are used to model serial I/O devices, such as keyboards and printers. Opening and reading from */dev/tty* reads from the keyboard; opening and writing to */dev/lp* writes to the printer. Block special files, often with names like */dev/hd1*, can be used to read and write raw disk partitions without regard to the file system. Thus, a seek to byte *k* followed by a read will begin reading from the *k*th byte on the corresponding partition, completely ignoring the i-node and file structure. Raw block devices are used for paging and swapping by programs that lay down file systems (e.g., *mkfs*), and by programs that fix sick file systems (e.g., *fsck*), for example.

Many computers have two or more disks. On mainframes at banks, for example, it is frequently necessary to have 100 or more disks on a single machine, in

order to hold the huge databases required. Personal computers may have an internal disk or SSD and an external USB drive for backups. When there are multiple disk drives, the question arises of how to handle them.

One solution is to put a self-contained file system on each one and just keep them separate. Consider, for example, the situation shown in Fig. 10-25(a). Here we have a hard disk, which we call *C*:, and a USB external drive, which we call *D*:. Each has its own root directory and files. With this solution, the user has to specify both the device and the file when anything other than the default is needed. For instance, to copy a file *x* to a directory *d* (assuming *C:* is the default), one would type

```
cp  D:/x  /a/d/x
```

This is the approach taken by a number of systems, including Windows, which it inherited from MS-DOS in a century long ago.

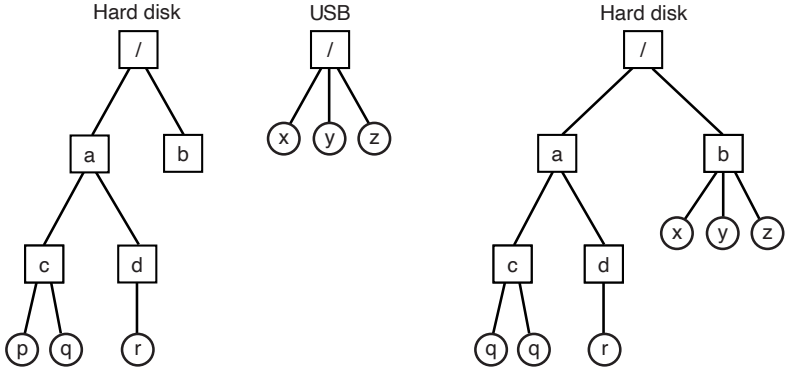

Figure 10-25. (a) Separate file systems. (b) After mounting.

The Linux solution is to allow one disk to be mounted in another disk's file tree. In our example, we could mount the USB drive on the directory */b*, yielding the file system of Fig. 10-25(b). The user now sees a single file tree, and no longer has to be aware of which file resides on which device. The above copy command now becomes

```
cp  /b/x  /a/d/x
```

exactly the same as it would have been if everything had been on the hard disk in the first place.

Another interesting property of the Linux file system is **locking**. In some applications, two or more processes may be using the same file at the same time, which may lead to race conditions. One solution is to program the application with critical regions. However, if the processes belong to independent users who do not even know each other, this kind of coordination is generally inconvenient.

Consider, for example, a database consisting of many files in one or more directories that are accessed by unrelated users. It is certainly possible to associate a semaphore with each directory or file and achieve mutual exclusion by having processes do a down operation on the appropriate semaphore before accessing the data. The disadvantage, however, is that a whole directory or file is then made inaccessible, even though only one record may be needed.

For this reason, POSIX provides a flexible and fine-grained mechanism for processes to lock as little as a single byte and as much as an entire file in one indivisible operation. The locking mechanism requires the caller to specify the file to be locked, the starting byte, and the number of bytes. If the operation succeeds, the system makes a table entry noting that the bytes in question (e.g., a database record) are locked.

Two kinds of locks are provided, **shared locks** and **exclusive locks**. If a portion of a file already contains a shared lock, a second attempt to place a shared lock on it is permitted, but an attempt to put an exclusive lock on it will fail. If a portion of a file contains an exclusive lock, all attempts to lock any part of that portion will fail until the lock has been released. In order to successfully place a lock, every byte in the region to be locked must be available.

When placing a lock, a process must specify whether it wants to block or not in the event that the lock cannot be placed. If it chooses to block, when the existing lock has been removed, the process is unblocked and the lock is placed. If the process chooses not to block when it cannot place a lock, the system call returns immediately, with the status code telling whether the lock succeeded or not. If it did not, the caller has to decide what to do next (e.g., wait and try again).

Locked regions may overlap. In Fig. 10-26(a) we see that process *A* has placed a shared lock on bytes 4 through 7 of some file. Later, process *B* places a shared lock on bytes 6 through 9, as shown in Fig. 10-26(b). Finally, *C* locks bytes 2 through 11. As long as all these locks are shared, they can coexist.

Now consider what happens if a process tries to acquire an exclusive lock to byte 9 of the file of Fig. 10-26(c), with a request to block if the lock fails. Since two previous locks cover this block, the caller will block and will remain blocked until both *B* and *C* release their locks.

10.6.2 File-System Calls in Linux

Many system calls relate to files and the file system. First we will look at the system calls that operate on individual files. Later we will examine those that involve directories or the file system as a whole. To create a new file, the creat call can be used. (When Ken Thompson was once asked what he would do differently if he had the chance to reinvent UNIX, he replied that he would spell creat as create this time.) parameters provide the name of the file and the protection mode. Thus

```
fd = creat("abc", mode);
```

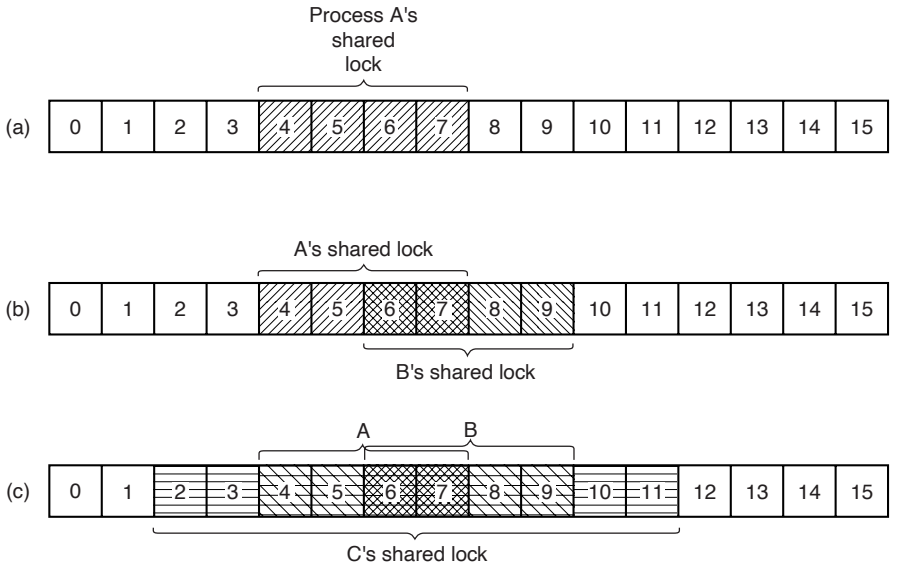

Figure 10-26. (a) A file with one lock. (b) Adding a second lock. (c) A third one.

creates a file called *abc* with the protection bits taken from *mode*. These bits determine which users may access the file and how. They will be described later.

The creat call not only creates a new file, but also opens it for writing. To allow subsequent system calls to access the file, a successful creat returns a small nonnegative integer called a **file descriptor**, *fd* in the example above. If a creat is done on an existing file, that file is truncated to length 0 and its contents are discarded. Additionally, files can also be created using the open call with appropriate arguments.

Now let us continue looking at the main file-system calls, which are listed in Fig. 10-27. To read or write an existing file, the file must first be opened by calling open or creat. This call specifies the file name to be opened and how it is to be opened: for reading, writing, or both. Various options can be specified as well. Like creat, the call to open returns a file descriptor that can be used for reading or writing. Afterward, the file can be closed by close, which makes the file descriptor available for reuse on a subsequent creat or open. Both the creat and open calls always return the lowest-numbered file descriptor not currently in use.

When a program starts executing in the standard way, file descriptors 0, 1, and 2 are already opened for standard input, standard output, and standard error, respectively. In this way, a filter, such as the *sort* program, can just read its input from file descriptor 0 and write its output to file descriptor 1, without having to know what files they are. This mechanism works because the shell arranges for these values to refer to the correct (redirected) files before the program is started.

System call	Description
fd = creat(name, mode)	One way to create a new file
fd = open(file, how, ...)	Open a file for reading, writing, or both
s = close(fd)	Close an open file
n = read(fd, buffer, nbytes)	Read data from a file into a buffer
n = write(fd, buffer, nbytes)	Write data from a buffer into a file
position = lseek(fd, offset, whence)	Move the file pointer
s = stat(name, &buf)	Get a file's status information
s = fstat(fd, &buf)	Get a file's status information
s = pipe(&fd[0])	Create a pipe
s = fcntl(fd, cmd, ...)	File locking and other operations

Figure 10-27. Some system calls relating to files. The return code *s* is −1 if an error has occurred; *fd* is a file descriptor, and *position* is a file offset. The parameters should be self-explanatory.

The most heavily used calls are undoubtedly read and write. Each one has three parameters: a file descriptor (telling which open file to read or write), a buffer address (telling where to put the data or get the data from), and a count (telling how many bytes to transfer). That is all there is. It is a very simple design. A typical call is

```
n = read(fd, buffer, nbytes);
```

Although nearly all programs read and write files sequentially, some programs need to be able to access any part of a file at random. Associated with each file is a pointer that indicates the current position in the file. When reading (or writing) sequentially, it normally points to the next byte to be read (written). If the pointer is at, say, 4096, before 1024 bytes are read, it will automatically be moved to 5120 after a successful read system call. The lseek call changes the value of the position pointer, so that subsequent calls to read or write can begin anywhere in the file, or even beyond the end of it. It is called lseek to avoid conflicting with seek, a now-obsolete call that was formerly used on 16-bit computers for seeking.

Lseek has three parameters: the first one is the file descriptor for the file; the second is a file position; the third tells whether the file position is relative to the beginning of the file, the current position, or the end of the file. The value returned by lseek is the absolute position in the file after the file pointer is changed. Slightly ironically, lseek is the only file system call that never causes a real disk operation because all it does is update the current file position, which is a number in memory.

For each file, Linux keeps track of the file mode (regular, directory, special file), size, time of last modification, and other information. Programs can ask to see

this information via the stat system call. The first parameter is the file name. The second is a pointer to a structure where the information requested is to be put. The fields in the structure are shown in Fig. 10-28. The fstat call is the same as stat except that it operates on an open file (whose name may not be known) rather than on a path name.

Device the file is on
I-node number (which file on the device)
File mode (includes protection information)
Number of links to the file
Identity of the file's owner
Group the file belongs to
File size (in bytes)
Creation time
Time of last access
Time of last modification

Figure 10-28. The fields returned by the stat system call.

The pipe system call is used to create shell pipelines. It creates a kind of pseudofile, which buffers the data between the pipeline components, and returns file descriptors for both reading and writing the buffer. In a pipeline such as

```
sort <in | head –30
```

file descriptor 1 (standard output) in the process running *sort* would be set (by the shell) to write to the pipe, and file descriptor 0 (standard input) in the process running *head* would be set to read from the pipe. In this way, *sort* just reads from file descriptor 0 (set to the file *in*) and writes to file descriptor 1 (the pipe) without even being aware that these have been redirected. If they have not been redirected, *sort* will automatically read from the keyboard and write to the screen (the default devices). Similarly, when *head* reads from file descriptor 0, it is reading the data *sort* put into the pipe buffer without even knowing that a pipe is in use. This is a clear example of how a simple concept (redirection) with a simple implementation (file descriptors 0 and 1) can lead to a powerful tool (connecting programs in arbitrary ways without having to modify them at all).

The last system call in Fig. 10-27 is fcntl. It is used to lock and unlock files, apply shared or exclusive locks, and perform a few other file-specific operations.

Now let us look at some system calls that relate more to directories or the file system as a whole, rather than just to one specific file. Some common ones are listed in Fig. 10-29. Directories are created and destroyed using mkdir and rmdir, respectively. A directory can be removed only if it is empty.

System call	Description
s = mkdir(path, mode)	Create a new directory
s = rmdir(path)	Remove a directory
s = link(oldpath, newpath)	Create a link to an existing file
s = unlink(path)	Unlink a file
s = chdir(path)	Change the working directory
dir = opendir(path)	Open a directory for reading
s = closedir(dir)	Close a directory
dirent = readdir(dir)	Read one directory entry
rewinddir(dir)	Rewind a directory so it can be reread

Figure 10-29. Some system calls relating to directories. The return code *s* is −1 if an error has occurred; *dir* identifies a directory stream, and *dirent* is a directory entry. The parameters should be self-explanatory.

As we saw in Fig. 10-24, linking to a file creates a new directory entry that points to an existing file. The link system call creates the link. The parameters specify the original and new names, respectively. Directory entries are removed with unlink. When the last link to a file is removed, the file is automatically deleted. For a file that has never been linked, the first unlink causes it to disappear.

The working directory is changed by the chdir system call. Doing so has the effect of changing the interpretation of relative path names.

The last four calls of Fig. 10-29 are for reading directories. They can be opened, closed, and read, analogous to ordinary files. Each call to readdir returns exactly one directory entry in a fixed format. There is no way for users to write in a directory (in order to maintain the integrity of the file system). Files can be added to a directory using creat or link and removed using unlink. There is also no way to seek to a specific file in a directory, but rewinddir allows an open directory to be read again from the beginning.

10.6.3 Implementation of the Linux File System

In this section, we will first look at the abstractions supported by the Virtual File System layer. The VFS hides from higher-level processes and applications the differences among many types of file systems supported by Linux, whether they are residing on local devices or are stored remotely and need to be accessed over the network. Devices and other special files are also accessed through the VFS layer. Next, we will describe the implementation of the first widespread Linux file system, ext2, or the second extended file system. Afterward, we will discuss the improvements in the ext4 file system. A wide variety of other file systems are also in use. All Linux systems can handle multiple disk partitions, each with a different file system on it.

The Linux Virtual File System

In order to enable applications to interact with different file systems, implemented on different types of local or remote devices, Linux takes an approach used in other UNIX systems: the Virtual File System (VFS). VFS defines a set of basic file-system abstractions and the operations which are allowed on these abstractions. Invocations of the system calls described in the previous section access the VFS data structures, determine the exact file system where the accessed file belongs, and via function pointers stored in the VFS data structures invoke the corresponding operation in the specified file system.

Figure 10-30 summarizes the four main file-system structures supported by VFS. The **superblock** contains critical information about the layout of the file system. Destruction of the superblock will render the file system unreadable. The **i-nodes** (short for index-nodes, but never called that, although some lazy people drop the hyphen and call them **inodes**) each describe exactly one file. Note that in Linux, directories and devices are also represented as files, thus they will have corresponding i-nodes. Both superblocks and i-nodes have a corresponding structure maintained on the physical disk where the file system resides.

Object	Description	Operation
Superblock	specific file-system	read_inode, sync_fs
Dentry	directory entry, single component of a path	d_compare, d_delete
I-node	specific file	create, link
File	open file associated with a process	read, write

Figure 10-30. File-system abstractions supported by the VFS.

In order to facilitate certain directory operations and traversals of paths, such as */usr/ast/bin*, VFS supports a **dentry** data structure which represents a directory entry. This data structure is created by the file system on the fly. Directory entries are cached in what is called the *dentry_cache*. For instance, the *dentry_cache* would contain entries for /, */usr*, */usr/ast*, and the like. If multiple processes access the same file through the same hard link (i.e., same path), their file object will point to the same entry in this cache.

Finally, the **file** data structure is an in-memory representation of an open file, and is created in response to the open system call. It supports operations such as read, write, sendfile, lock, and other system calls described in the previous section.

The actual file systems implemented underneath the VFS need not use the exact same abstractions and operations internally. They must, however, implement file-system operations semantically equivalent to those specified with the VFS objects. The elements of the *operations* data structures for each of the four VFS objects are pointers to functions in the underlying file system.

The Linux Ext2 File System

We next describe one of the earlier on-disk file systems used in Linux: **ext2**. The first Linux release used the MINIX 1 file system and was limited by short file names (chosen for UNIX V7 compatibility) and 64-MB file sizes. The MINIX 1 file system was eventually replaced by the first extended file system, **ext**, which permitted both longer file names and larger file sizes. Due to its performance inefficiencies, ext was replaced by its successor, ext2, which reached widespread use.

An ext2 Linux disk partition contains a file system with the layout shown in Fig. 10-31. Block 0 is not used by Linux and contains code to boot the computer. Following block 0, the disk partition is divided into groups of blocks, irrespective of where the disk cylinder boundaries fall. Each group is organized as follows.

The first block is the superblock. It contains information about the layout of the file system, including the number of i-nodes, the number of disk blocks, and the start of the list of free disk blocks (typically a few hundred entries). Next comes the group descriptor, which contains information about the location of the bitmaps, the number of free blocks and i-nodes in the group, and the number of directories in the group. This information is important since ext2 attempts to spread directories evenly over the disk.

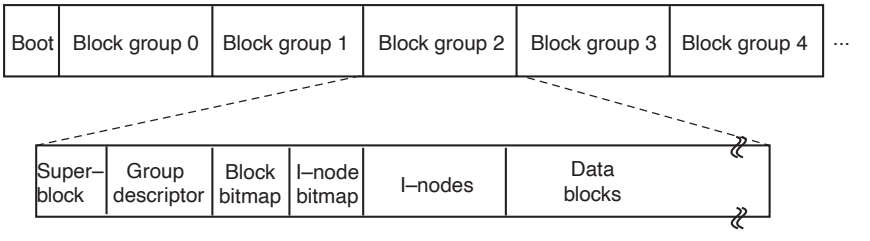

Figure 10-31. Disk layout of the Linux ext2 file system.

Two bitmaps are used to keep track of the free blocks and free i-nodes, respectively, a choice inherited from the MINIX 1 file system (and in contrast to most UNIX file systems, which use a free list). Each map is one block long. With a 1-KB block, this design limits a block group to 8192 blocks and 8192 i-nodes. The former is a real restriction but, in practice, the latter is not. With 4-KB blocks, the numbers are four times larger.

Following the superblock are the i-nodes themselves. They are numbered from 1 up to some maximum. Each i-node is 128 bytes long and describes exactly one file. An i-node contains accounting information (including all the information returned by stat, which simply takes it from the i-node), as well as enough information to locate all the disk blocks that hold the file's data.

Following the i-nodes are the data blocks. All the files and directories are stored here. If a file or directory consists of more than one block, the blocks need not

be contiguous on the disk. In fact, the blocks of a large file are likely to be spread all over the disk.

I-nodes corresponding to directories are dispersed throughout the disk block groups. Ext2 makes an effort to collocate ordinary files in the same block group as the parent directory, and data files in the same block as the original file i-node, provided that there is sufficient space. This idea was borrowed from the Berkeley Fast File System (McKusick et al., 1984). The bitmaps are used to make quick decisions regarding where to allocate new file-system data. When new file blocks are allocated, ext2 also *preallocates* a number (eight) of additional blocks for that file, so as to minimize the file fragmentation due to future write operations. This scheme balances the file-system load across the entire disk. It also performs well due to its tendencies for collocation and reduced fragmentation.

To access a file, it must first use one of the Linux system calls, such as open, which requires the file's path name. The path name is parsed to extract individual directories. If a relative path is specified, the lookup starts from the process' current directory, otherwise it starts from the root directory. In either case, the i-node for the first directory can easily be located: there is a pointer to it in the process descriptor, or, in the case of a root directory, it is typically stored in a predetermined block on disk.

The directory file allows file names up to 255 characters and is illustrated in Fig. 10-32. Each directory consists of some integral number of disk blocks so that directories can be written atomically to the disk. Within a directory, entries for files and directories are in unsorted order, with each entry directly following the one before it. Entries may not span disk blocks, so often there are some number of unused bytes at the end of each disk block.

Each directory entry in Fig. 10-32 consists of four fixed-length fields and one variable-length field. The first field is the i-node number, 19 for the file *colossal*, 42 for the file *voluminous*, and 88 for the directory *bigdir*. Next comes a field rec_len, telling how big the entry is (in bytes), possibly including some padding after the name. This field is needed to find the next entry for the case that the file name is padded by an unknown length. That is the meaning of the arrow in Fig. 10-32. Then comes the type field: file, directory, and so on. The last fixed field is the length of the actual file name in bytes, 8, 10, and 6 in this example. Finally, comes the file name itself, terminated by a 0 byte and padded out to a 32-bit boundary. Additional padding may follow that.

In Fig. 10-32(b) we can see the same directory after the entry for *voluminous* has been removed from the directory. All the removal has done is increase the size of the total entry field for *colossal*, turning the former field for *voluminous* into padding for the first entry. This padding can be used for a subsequent entry, of course.

Since directories are searched linearly, it can take a long time to find an entry at the end of a large directory. Therefore, the system maintains a cache of recently accessed directories. This cache is searched using the name of the file, and if a hit

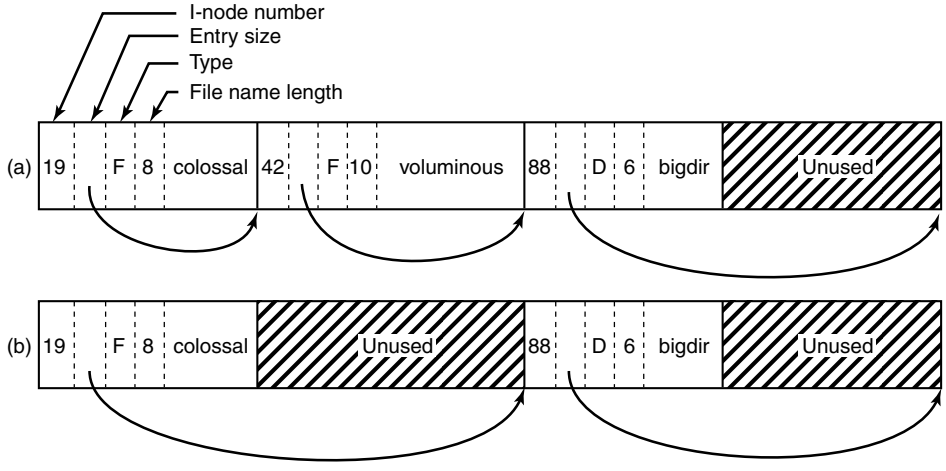

Figure 10-32. (a) A Linux directory with three files. (b) The same directory after the file *voluminous* has been removed.

occurs, the costly linear search is avoided. A *dentry* object is entered in the dentry cache for each of the path components, and, through its i-node, the directory is searched for the subsequent path element entry, until the actual file i-node is reached.

For instance, to look up a file specified with an absolute path name, such as */usr/ast/file*, the following steps are required. First, the system locates the root directory, which generally uses i-node 2, especially when i-node 1 is reserved for bad-block handling. It places an entry in the dentry cache for future lookups of the root directory. Then it looks up the string "usr" in the root directory, to get the i-node number of the */usr* directory, which is also entered in the dentry cache. This i-node is then fetched, and the disk blocks are extracted from it, so the */usr* directory can be read and searched for the string "ast". Once this entry is found, the i-node number for the */usr/ast* directory can be taken from it. Armed with the i-node number of the */usr/ast* directory, this i-node can be read and the directory blocks located. Finally, "file" is looked up and its i-node number found. Thus, the use of a relative path name is not only more convenient for the user, but it also saves a substantial amount of work for the system.

If the file is present, the system extracts the i-node number and uses it as an index into the i-node table (on disk) to locate the corresponding i-node and bring it into memory. The i-node is put in the **i-node table**, a kernel data structure that holds all the i-nodes for currently open files and directories. The format of the i-node entries, as a bare minimum, must contain all the fields returned by the stat system call so as to make stat work (see Fig. 10-28). In Fig. 10-33 we show some of the fields included in the i-node structure supported by the Linux file-system

layer. The actual i-node structure contains quite a few more fields, since the same structure is also used to represent directories, devices, and other special files. The i-node structure also contains fields reserved for future use. History has shown that unused bits do not remain that way for long.

Field	Bytes	Description
Mode	2	File type, protection bits, setuid, setgid bits
Nlinks	2	Number of directory entries pointing to this i-node
Uid	2	UID of the file owner
Gid	2	GID of the file owner
Size	4	File size in bytes
Addr	60	Address of first 12 disk blocks, then 3 indirect blocks
Gen	1	Generation number (incremented every time i-node is reused)
Atime	4	Time the file was last accessed
Mtime	4	Time the file was last modified
Ctime	4	Time the i-node was last changed (except the other times)

Figure 10-33. Some fields in the i-node structure in Linux.

Let us now see how the system reads a file. Remember that a typical call to the library procedure for invoking the read system call looks like this:

```
n = read(fd, buffer, nbytes);
```

When the kernel gets control, all it has to start with are these three parameters and the information in its internal tables relating to the user. One of the items in the internal tables is the file-descriptor array. It is indexed by a file descriptor and contains one entry for each open file (up to the maximum number, often 32).

The idea is to start with this file descriptor and end up with the corresponding i-node. Let us consider one possible design: just put a pointer to the i-node in the file-descriptor table. Although simple, unfortunately this method does not work. The problem is as follows. Associated with every file descriptor is a file position that tells at which byte the next read (or write) will start. Where should it go? One possibility is to put it in the i-node table. However, this approach fails if two or more unrelated processes happen to open the same file at the same time because each one has its own file position.

A second possibility is to put the file position in the file-descriptor table. In that way, every process that opens a file gets its own private file position. Unfortunately this scheme fails too, but the reasoning is more subtle and has to do with the nature of file sharing in Linux. Consider a shell script, *s*, consisting of two commands, *p1* and *p2*, to be run in order. If the shell script is called by the command

```
s >x
```

it is expected that *p1* will write its output to *x*, and then *p2* will write its output to *x* also, starting at the place where *p1* stopped.

When the shell forks off *p1*, *x* is initially empty, so *p1* just starts writing at file position 0. However, when *p1* finishes, some mechanism is needed to make sure that the initial file position that *p2* sees is not 0 (which it would be if the file position were kept in the file-descriptor table), but the value *p1* ended with.

The way this is achieved is shown in Fig. 10-34. The trick is to introduce a new table, the **open-file-description table**, between the file descriptor table and the i-node table, and put the file position (and read/write bit) there. In this figure, the parent is the shell and the child is first *p1* and later *p2*. When the shell forks off *p1*, its user structure (including the file-descriptor table) is an exact copy of the shell's, so both of them point to the same open-file-description table entry. When *p1* finishes, the shell's file descriptor is still pointing to the open-file description containing *p1*'s file position. When the shell now forks off *p2*, the new child automatically inherits the file position, without either it or the shell even having to know what that position is.

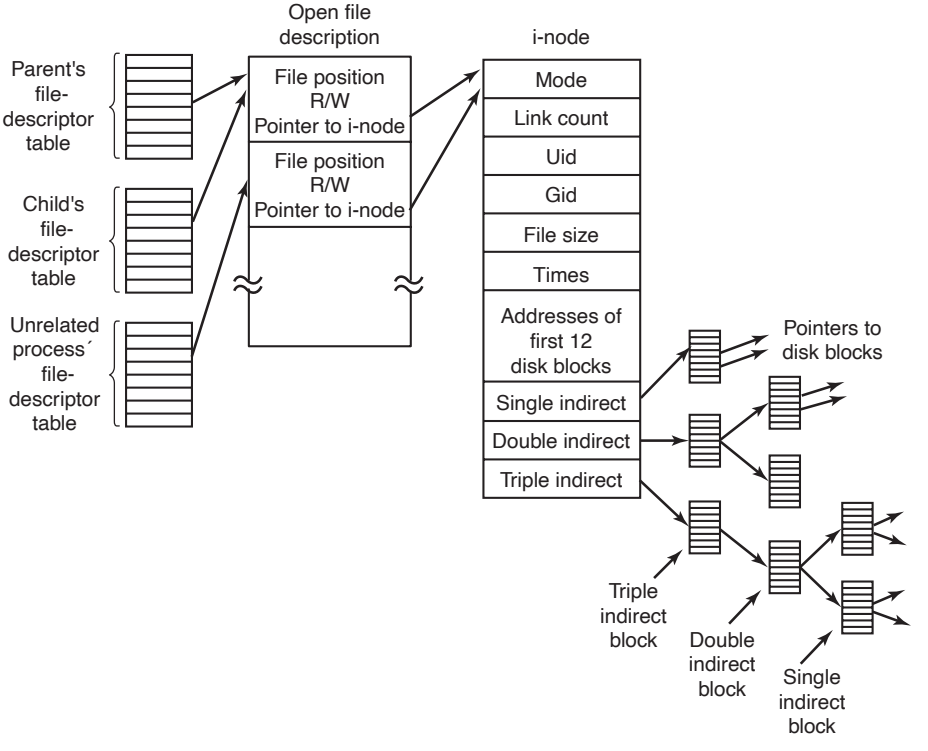

Figure 10-34. The relation between the file-descriptor table, the open-file-description table, and the i-node table.

However, if an unrelated process opens the file, it gets its own open-file-description entry, along with its own file position, which is just what is needed.

Thus the whole point of the open-file-description table is to allow a parent and child to share a file position, but to provide unrelated processes with their own values.

Getting back to the problem of doing the read, we have now shown how the file position and i-node are located. The i-node contains the disk addresses of the first 12 blocks of the file. If the file position falls in the first 12 blocks, the block is read and the data are copied to the user. For files longer than 12 blocks, a field in the i-node contains the disk address of a **single indirect block**, as shown in Fig. 10-34. This block contains the disk addresses of more disk blocks. For example, if a block is 1 KB and a disk address is 4 bytes, the single indirect block can hold 256 disk addresses. Thus this scheme works for files of up to 268 KB.

Beyond that, a **double indirect block** is used. It contains the addresses of 256 single indirect blocks, each of which holds the addresses of 256 data blocks. This mechanism is sufficient to handle files up to $10 + 2^{16}$ blocks (67,119,104 bytes). If even this is not enough, the i-node has space for a **triple indirect block**. Its pointers point to many double indirect blocks. This addressing scheme can handle file sizes of 2^{24} 1-KB blocks (16 GB). For 8-KB block sizes, the addressing scheme can support file sizes up to 64 TB.

The Linux Ext4 File System

In order to prevent all data loss after system crashes and power failures, the ext2 file system would have to write out each data block to disk as soon as it was created. The latency incurred during the required disk-head seek operation would be so high that the performance would be intolerable. Therefore, writes are delayed, and changes may not be committed to disk for up to 30 sec, which is a very long time interval in the context of modern computer hardware.

To improve the robustness of the file system, Linux relies on **journaling file systems**. **Ext3**, a successor of the ext2 file system, is an example of a journaling file system. **Ext4**, a follow-on of ext3, is also a journaling file system, but unlike ext3, it changes the block addressing scheme used by its predecessors, thereby supporting both larger files and larger overall file-system sizes. Today, it is considered to be the most popular among the Linux file systems. We will describe some of its features next.

The basic idea behind a journaling file system is to maintain a *journal*, which describes all file-system operations in sequential order. By sequentially writing out changes to the file-system data or metadata (i-nodes, superblock, etc.), the operations do not suffer from the overheads of disk-head movement during random disk accesses. Eventually, the changes will be written out, committed, to the appropriate disk location, and the corresponding journal entries can be discarded. If a system crash or power failure occurs before the changes are committed, during restart the system will detect that the file system was not unmounted properly, traverse the journal, and apply the file-system changes described in the journal log.

Ext4 is designed to be highly compatible with ext2 and ext3, although its core data structures and disk layout are modified. Regardless, a file system which has been unmounted as an ext2 system can be subsequently mounted as an ext4 system and offer the journaling capability.

The journal is a file managed as a circular buffer. The journal may be stored on the same or a separate device from the main file system. Since the journal operations are not "journaled" themselves, these are not handled by the same ext4 file system. Instead, a separate **JBD** (**Journaling Block Device**) is used to perform the journal read/write operations.

JBD supports three main data structures: *log record*, *atomic operation handle*, and *transaction*. A log record describes a low-level file-system operation, typically resulting in changes within a block. Since a system call such as write includes changes at multiple places—i-nodes, existing file blocks, new file blocks, list of free blocks, etc.—related log records are grouped in atomic operations. Ext4 notifies JBD of the start and end of system-call processing, so that JBD can ensure that either all log records in an atomic operation are applied, or none of them. Finally, primarily for efficiency reasons, JBD treats collections of atomic operations as transactions. Log records are stored consecutively within a transaction. JBD will allow portions of the journal file to be discarded only after all log records belonging to a transaction are safely committed to disk.

Since writing out a log entry for each disk change may be costly, ext4 may be configured to keep a journal of all disk changes, or only of changes related to the file-system metadata (the i-nodes, superblocks, etc.). Journaling only metadata gives less system overhead and results in better performance but does not make any guarantees against corruption of file data. Several other journaling file systems maintain logs of only metadata operations (e.g., SGI's XFS). In addition, the reliability of the journal can be further improved via checksumming.

Key modification in ext4 compared to its predecessors is the use of **extents**. Extents represent contiguous blocks of storage, for instance 128 MB of contiguous 4-KB blocks vs. individual storage blocks, as referenced in ext2. Unlike its predecessors, ext4 does not require metadata operations for each block of storage. This scheme also reduces fragmentation for large files. As a result, ext4 can provide faster file system operations and support larger files and file system sizes. For instance, for a block size of 1 KB, ext4 increases the maximum file size from 16 GB to 16 TB, and the maximum file system size to 1 EB (Exabyte).

The /proc File System

Another Linux file system is the **/proc** (process) file system, an idea originally devised in the 8th edition of UNIX from Bell Labs and later copied in 4.4BSD and System V. However, Linux extends the idea in several ways. The basic concept is that for every process in the system, a directory is created in */proc*. The name of the directory is the process PID expressed as a decimal number. For example,

/proc/619 is the directory corresponding to the process with PID 619. In this directory are files that appear to contain information about the process, such as its command line, environment strings, and signal masks. In fact, these files do not exist on the disk. When they are read, the system retrieves the information from the actual process as needed and returns it in a standard format.

Many of the Linux extensions relate to other files and directories located in */proc*. They contain a wide variety of information about the CPU, disk partitions, devices, interrupt vectors, kernel counters, file systems, loaded modules, and much more. Unprivileged user programs may read much of this information to learn about system behavior in a safe way. Some of these files may be written to in order to change system parameters.

10.6.4 NFS: The Network File System

Networking has played a major role in Linux, and UNIX in general, right from the beginning (the first UNIX network was built to move new kernels from the PDP-11/70 to the Interdata 8/32 during the port to the latter). In this section, we will examine Sun Microsystem's **NFS** (**Network File System**), which is used on all modern Linux systems to join the file systems on separate computers into one logical whole. NSF version 3 was introduced in 1994. Currently, the most recent version is NSFv4. It was originally introduced in 2000 and provides several enhancements over the previous NFS architecture. Three aspects of NFS are of interest: the architecture, the protocol, and the implementation. We will now examine these in turn, first in the context of the simpler NFS version 3, then we will turn to the enhancements included in v4.

NFS Architecture

The basic idea behind NFS is to allow an arbitrary collection of clients and servers to share a common file system. In many cases, all the clients and servers are on the same LAN, but this is not required. It is also possible to run NFS over a wide area network if the server is far from the client. For simplicity, we will speak of clients and servers as though they were on distinct machines, but in fact, NFS allows every machine to be both a client and a server at the same time.

Each NFS server exports one or more of its directories for access by remote clients. When a directory is made available, so are all of its subdirectories, so actually entire directory trees are normally exported as a unit. The list of directories a server exports is maintained in a file, often */etc/exports*, so these directories can be exported automatically whenever the server is booted. Clients access exported directories by mounting them. When a client mounts a (remote) directory, it becomes part of its directory hierarchy, as shown in Fig. 10-35.

In this example, client 1 has mounted the *bin* directory of server 1 on its own *bin* directory, so it can now refer to the shell as */bin/sh* and get the shell on server

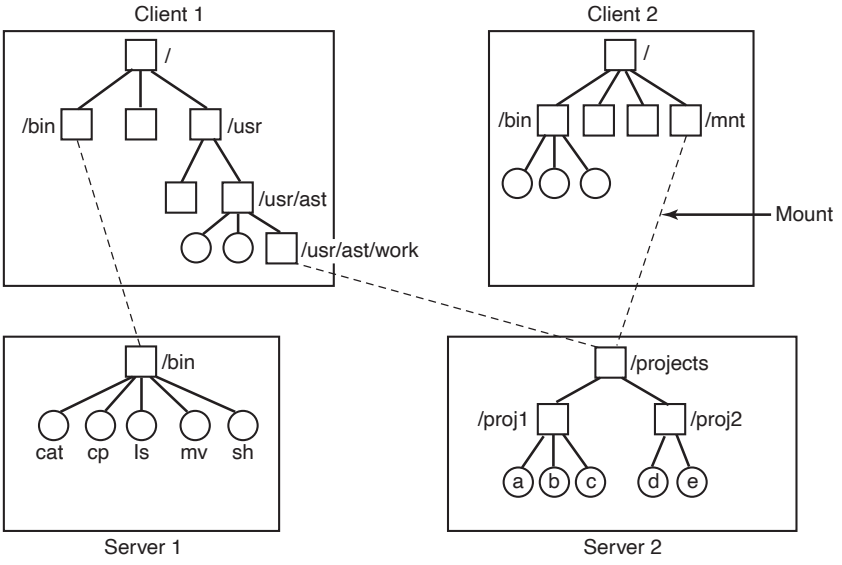

Figure 10-35. Examples of remote mounted file systems. Directories are shown as squares and files as circles.

1. Diskless workstations often have only a skeleton file system (in RAM) and get all their files from remote servers like this. Similarly, client 1 has mounted server 2's directory */projects* on its directory */usr/ast/work* so it can now access file *a* as */usr/ast/work/proj1/a*. Finally, client 2 has also mounted the *projects* directory and can also access file *a*, only as */mnt/proj1/a*. As seen here, the same file can have different names on different clients due to its being mounted in a different place in the respective trees. The mount point is entirely local to the clients; the server does not know where it is mounted on any of its clients.

NFS Protocols

Since one of the goals of NFS is to support a heterogeneous system, with clients and servers possibly running different operating systems on different hardware, it is essential that the interface between the clients and servers be well defined. Only then is anyone able to write a new client implementation and expect it to work correctly with existing servers, and vice versa.

NFS accomplishes this goal by defining two client-server protocols. A **protocol** is a set of requests sent by clients to servers, along with the corresponding replies sent by the servers back to the clients.

The first NFS protocol handles mounting. A client can send a path name to a server and request permission to mount that directory somewhere in its directory hierarchy. The place where it is to be mounted is not contained in the message, as the server does not care where it is to be mounted. If the path name is legal and the directory specified has been exported, the server returns a **file handle** to the client. The file handle contains fields uniquely identifying the file-system type, the disk, the i-node number of the directory, and security information. Subsequent calls to read and write files in the mounted directory or any of its subdirectories use the file handle. It is somewhat analogous to the file descriptors returned by the creat and open calls on local files.

When Linux boots, it runs the */etc/rc* shell script before going multiuser. Commands to mount remote file systems can be placed in this script, thus automatically mounting the necessary remote file systems before allowing any logins. Alternatively, most versions of Linux also support **automounting**. This feature allows a set of remote directories to be associated with a local directory. None of these remote directories are mounted (or their servers even contacted) when the client is booted. Instead, the first time a remote file is opened, the operating system sends a message to each of the servers. The first one to reply wins, and its directory is mounted.

Automounting has two principal advantages over static mounting via the */etc/rc* file. First, if one of the NFS servers named in */etc/rc* happens to be down, it is impossible to bring the client up, at least not without some difficulty, delay, and quite a few error messages. If the user does not even need that server at the moment, all that work is wasted. Second, by allowing the client to try a set of servers in parallel, a degree of fault tolerance can be achieved (because only one of them needs to be up), and the performance can be improved (by choosing the first one to reply—presumably the least heavily loaded).

On the other hand, it is tacitly assumed that all the file systems specified as alternatives for the automount are identical. Since NFS provides no support for file or directory replication, it is up to the user to arrange for all the file systems to be the same. Consequently, automounting is most often used for read-only file systems containing system binaries and other files that rarely change.

The second NFS protocol is for directory and file access. Clients can send messages to servers to manipulate directories and read and write files. They can also access file attributes, such as file mode, size, and time of last modification. Most Linux system calls are supported by NFS, with the perhaps surprising exceptions of open and close.

The omission of open and close is not an accident. It is fully intentional. It is not necessary to open a file before reading it, nor to close it when done. Instead, to read a file, a client sends the server a lookup message containing the file name, with a request to look it up and return a file handle, which is a structure that identifies the file (i.e., contains a file system identifier and i-node number, among other data). Unlike an open call, this lookup operation does not copy any information

into internal system tables. The read call contains the file handle of the file to read, the offset in the file to begin reading, and the number of bytes desired. Each such message is self-contained. The advantage of this scheme is that the server does not have to remember anything about open connections in between calls to it. Thus if a server crashes and then recovers, no information about open files is lost, because there is none. A server like this that does not maintain state information about open files is said to be **stateless**.

Unfortunately, the NFS method makes it difficult to achieve the exact Linux file semantics. For example, in Linux a file can be opened and locked so that other processes cannot access it. When the file is closed, the locks are released. In a stateless server such as NFS, locks cannot be associated with open files, because the server does not know which files are open. NFS therefore needs a separate, additional mechanism to handle locking.

NFS uses the standard UNIX protection mechanism, with the *rwx* bits for the owner, group, and others (mentioned in Chap. 1 and discussed in detail below). Originally, each request message simply contained the user and group IDs of the caller, which the NFS server used to validate the access. In effect, it trusted the clients not to cheat. Several years' experience abundantly demonstrated that such an assumption was—how shall we put it?—rather naive. Currently, public key cryptography can be used to establish a secure key for validating the client and server on each request and reply. When this option is used, a malicious client cannot impersonate another client because it does not know that client's secret key.

NFS Implementation

Although the implementation of the client and server code is independent of the NFS protocols, most Linux systems use a three-layer implementation similar to that of Fig. 10-36. The top layer is the system-call layer. This handles calls like open, read, and close. After parsing the call and checking the parameters, it invokes the second layer, the Virtual File System (VFS) layer.

The task of the VFS layer is to maintain a table with one entry for each open file. The VFS layer additionally has an entry, a **virtual i-node**, or **v-node**, for every open file. The term v-node comes from BSD. In Linux, v-nodes are (confusingly) referred to as generic i-nodes, in contrast to the file system-specific i-nodes stored on disk. V-nodes are used to tell whether the file is local or remote. For remote files, enough information is provided to be able to access them. For local files, the file system and i-node are recorded because modern Linux systems can support multiple file systems (e.g., ext4fs, /proc, XFS, etc.). Although VFS was invented to support NFS, most modern Linux systems now support it as an integral part of the operating system, even if NFS is not used.

To see how v-nodes are used, let us trace a sequence of mount, open, and read system calls. To mount a remote file system, the system administrator (or */etc/rc*) calls the *mount* program specifying the remote directory, the local directory on

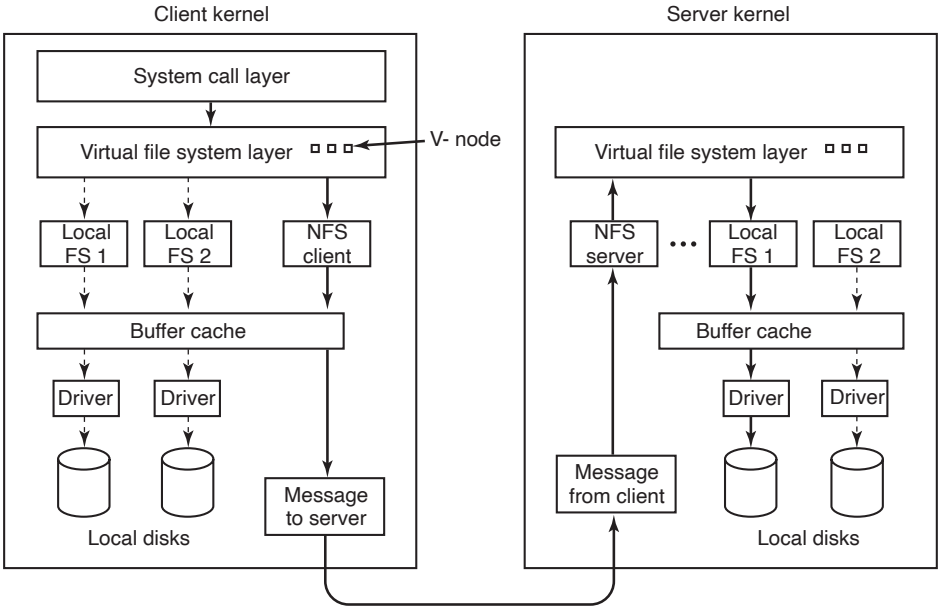

Figure 10-36. The NFS layer structure.

which it is to be mounted, and other information. The *mount* program parses the name of the remote directory to be mounted and discovers the name of the NFS server on which the remote directory is located. It then contacts that machine, asking for a file handle for the remote directory. If the directory exists and is available for remote mounting, the server returns a file handle for the directory. Finally, it makes a mount system call, passing the handle to the kernel.

The kernel then constructs a v-node for the remote directory and asks the NFS client code in Fig. 10-36 to create an **r-node** (**remote i-node**) in its internal tables to hold the file handle. The v-node points to the r-node. Each v-node in the VFS layer will ultimately contain either a pointer to an r-node in the NFS client code, or a pointer to an i-node in one of the local file systems (shown as dashed lines in Fig. 10-36). Thus, from the v-node it is possible to see if a file or directory is local or remote. If it is local, the correct file system and i-node can be located. If it is remote, the remote host and file handle can be located.

When a remote file is opened on the client, at some point during the parsing of the path name, the kernel hits the directory on which the remote file system is mounted. It sees that this directory is remote and in the directory's v-node finds the pointer to the r-node. It then asks the NFS client code to open the file. The NFS client code looks up the remaining portion of the path name on the remote server associated with the mounted directory and gets back a file handle for it. It makes an r-node for the remote file in its tables and reports back to the VFS layer,

which puts in its tables a v-node for the file that points to the r-node. Again here we see that every open file or directory has a v-node that points to either an r-node or an i-node.

The caller is given a file descriptor for the remote file. This file descriptor is mapped onto the v-node by tables in the VFS layer. Note that no table entries are made on the server side. Although the server is prepared to provide file handles upon request, it does not keep track of which files happen to have file handles outstanding and which do not. When a file handle is sent to it for file access, it checks the handle, and if it is valid, uses it. Validation can include verifying an authentication key contained in the RPC headers, if security is enabled.

When the file descriptor is used in a subsequent system call, for example, read, the VFS layer locates the corresponding v-node, and from that determines whether it is local or remote and also which i-node or r-node describes it. It then sends a message to the server containing the handle, the file offset (which is maintained on the client side, not the server side), and the byte count. For efficiency reasons, transfers between client and server are done in large chunks, normally 8192 bytes, even if fewer bytes are requested. The chunk size is configurable, up to a limit, and must be a multiple of 4 KB.

When the request message arrives at the server, it is passed to the VFS layer there, which determines which local file system holds the requested file. The VFS layer then makes a call to that local file system to read and return the bytes. These data are then passed back to the client. After the client's VFS layer has gotten the 8-KB chunk it asked for, it automatically issues a request for the next chunk, so it will have it should it be needed shortly. This feature, known as **read ahead**, improves performance considerably.

For writes, an analogous path is followed from client to server. Also, transfers are done in 8-KB chunks here, too. If a write system call supplies fewer than 8 KB of data, the data are just accumulated locally. Only when the entire 8-KB chunk is full is it sent to the server. However, when a file is closed, all of its data are sent to the server immediately.

Another technique used to improve performance is caching, as in ordinary UNIX. Servers cache data to avoid disk accesses, but this is invisible to the clients. Clients maintain two caches, one for file attributes (i-nodes) and one for file data. When either an i-node or a file block is needed, a check is made to see if it can be satisfied out of the cache. If so, network traffic can be avoided.

While client caching helps performance enormously, it also introduces some nasty problems. Suppose that two clients are both caching the same file block and one of them modifies it. When the other one reads the block, it gets the old (stale) value. The cache is not coherent.

Given the potential severity of this problem, the NFS implementation does several things to mitigate it. For one, associated with each cache block is a timer. When the timer expires, the entry is discarded. Normally, the timer is 3 sec for data blocks and 30 sec for directory blocks. Doing this reduces the risk somewhat. In

addition, whenever a cached file is opened, a message is sent to the server to find out when the file was last modified. If the last modification occurred after the local copy was cached, the cache copy is discarded and the new copy fetched from the server. Finally, once every 30 sec a cache timer expires, and all the dirty (i.e., modified) blocks in the cache are sent to the server. While not perfect, these patches make the system highly usable in most practical circumstances.

NFS Version 4

Version 4 of the Network File System was designed to simplify certain operations from its predecessor. In contrast to NSFv3, which is described above, NFSv4 is a **stateful** file system. This permits `open` operations to be invoked on remote files, since the remote NFS server will maintain all file-system-related structures, including the file pointer. Read operations then need not include absolute read ranges, but can be incrementally applied from the previous file-pointer position. This results in shorter messages, and also in the ability to bundle multiple NFSv3 operations in one network transaction.

The stateful nature of NFSv4 makes it easy to integrate the variety of NFSv3 protocols described earlier in this section into one coherent protocol. There is no need to support separate protocols for mounting, caching, locking, or secure operations. NFSv4 also works better with both Linux (and UNIX in general) and Windows file-system semantics.

10.7 SECURITY IN LINUX

Linux, as a clone of MINIX and UNIX, has been a multiuser system almost from the beginning. This history means that security and control of information was built in very early on. In the following sections, we will look at some of the security aspects of Linux.

10.7.1 Fundamental Concepts

The user community for a Linux system consists of some number of registered users, each of whom has a unique **UID** (**User ID**). A UID is an integer between 0 and 65,535. Files (but also processes and other resources) are marked with the UID of their owner. By default, the owner of a file is the person who created the file, although there is a way to change ownership.

Users can be organized into groups, which are also numbered with 16-bit integers called **GIDs** (**Group IDs**). Assigning users to groups is done manually (by the system administrator) and consists of making entries in a system database telling which user is in which group. A user could be in one or more groups at the same time. For simplicity, we will not discuss this feature further.

The basic security mechanism in Linux is simple. Each process carries the UID and GID of its owner. When a file is created, it gets the UID and GID of the creating process. The file also gets a set of permissions determined by the creating process. These permissions specify what access the owner, the other members of the owner's group, and the rest of the users have to the file. For each of these three categories, potential accesses are read, write, and execute, designated by the letters *r*, *w*, and *x*, respectively. The ability to execute a file makes sense only if that file is an executable binary program, of course. An attempt to execute a file that has execute permission but which is not executable (i.e., does not start with a valid header) will fail with an error. Since there are three categories of users and 3 bits per category, 9 bits are sufficient to represent the access rights. Some examples of these 9-bit numbers and their meanings are given in Fig. 10-37.

Binary	Symbolic	Allowed file accesses
111000000	rwx------	Owner can read, write, and execute
111111000	rwxrwx---	Owner and group can read, write, and execute
110100000	rw-r-----	Owner can read and write; group can read
110100100	rw-r--r--	Owner can read and write; all others can read
111101101	rwxr-xr-x	Owner can do everything, rest can read and execute
000000000	---------	Nobody has any access
000000111	------rwx	Only outsiders have access (strange, but legal)

Figure 10-37. Some example file-protection modes.

The first two entries in Fig. 10-37 allow the owner and the owner's group full access, respectively. The next one allows the owner's group to read the file but not to change it, and prevents outsiders from any access. The fourth entry is common for a data file the owner wants to make public. Similarly, the fifth entry is the usual one for a publicly available program. The sixth entry denies all access to all users. This mode is sometimes used for dummy files used for mutual exclusion because an attempt to create such a file will fail if one already exists. Thus if multiple processes simultaneously attempt to create such a file as a lock, only one of them will succeed. The last example is strange indeed, since it gives the rest of the world more access than the owner. However, its existence follows from the protection rules. Fortunately, there is a way for the owner to subsequently change the protection mode, even without having any access to the file itself.

The user with UID 0 is special and is called the **superuser** (or **root**). The superuser has the power to read and write all files in the system, no matter who owns them and no matter how they are protected. Processes with UID 0 also have the ability to make a small number of protected system calls denied to ordinary users. Normally, only the system administrator knows the superuser's password, although many undergraduates consider it a great sport to try to look for security flaws in the system so they can log in as the superuser without knowing the password. Management tends to frown on such activity.

Directories are files and have the same protection modes that ordinary files do except that the *x* bits refer to search permission instead of execute permission. Thus a directory with mode *rwxr–xr–x* allows its owner to read, modify, and search the directory, but allows others only to read and search it, but not add or remove files from it.

Special files corresponding to the I/O devices have the same protection bits as regular files. This mechanism can be used to limit access to I/O devices. For example, the printer special file, */dev/lp*, could be owned by the root or by a special user, daemon, and have mode *rw–––––––* to keep everyone else from directly accessing the printer. After all, if everyone could just print at will, chaos would result.

Of course, having */dev/lp* owned by, say, daemon with protection mode *rw–––––––* means that nobody else can use the printer. While this would save many innocent trees from an early death, sometimes users do have a legitimate need to print something. In fact, there is a more general problem of allowing controlled access to all I/O devices and other system resources.

This problem was solved by adding a new protection bit, the **SETUID bit**, to the 9 protection bits discussed above. When a program with the SETUID bit on is executed, the **effective UID** for that process becomes the UID of the executable file's owner instead of the UID of the user who invoked it. When a process attempts to open a file, it is the effective UID that is checked, not the underlying real UID. By making the program that accesses the printer be owned by daemon but with the SETUID bit on, any user could execute it, and have the power of daemon (e.g., access to */dev/lp*) but only to run that program (which might queue print jobs for printing in an orderly fashion).

Many sensitive Linux programs are owned by the root but with the SETUID bit on. For example, the program that allows users to change their passwords, *passwd*, needs to write in the password file. Making the password file publicly writable would not be a good idea. Instead, there is a program that is owned by the root and which has the SETUID bit on. Although the program has complete access to the password file, it will change only the caller's password and not permit any other access to the password file.

In addition to the SETUID bit, there is also a SETGID bit that works analogously, temporarily giving the user the effective GID of the program. In practice, this bit is rarely used, however.

10.7.2 Security System Calls in Linux

There are only a small number of system calls relating to security. The most important ones are listed in Fig. 10-38. The most heavily used security system call is chmod. It is used to change the protection mode. For example,

```
s = chmod("/usr/ast/newgame", 0755);
```

sets *newgame* to *rwxr–xr–x* so that everyone can run it (note that 0755 is an octal constant, which is convenient, since the protection bits come in groups of 3 bits). Only the owner of a file and the superuser can change its protection bits.

System call	Description
s = chmod(path, mode)	Change a file's protection mode
s = access(path, mode)	Check access using the real UID and GID
uid = getuid( )	Get the real UID
uid = geteuid( )	Get the effective UID
gid = getgid( )	Get the real GID
gid = getegid( )	Get the effective GID
s = chown(path, owner, group)	Change owner and group
s = setuid(uid)	Set the UID
s = setgid(gid)	Set the GID

Figure 10-38. Some system calls relating to security. The return code *s* is –1 if an error has occurred; *uid* and *gid* are the UID and GID, respectively. The parameters should be self-explanatory.

The access call tests to see if a particular access would be allowed using the real UID and GID. This system call is needed to avoid security breaches in programs that are SETUID and owned by the root. Such a program can do anything, and it is sometimes needed for the program to figure out if the user is allowed to perform a certain access. The program cannot just try it, because the access will always succeed. With the access call, the program can find out if the access is allowed by the real UID and real GID.

The next four system calls return the real and effective UIDs and GIDs. The last three are allowed only for the superuser. They change a file's owner, and a process' UID and GID.

10.7.3 Implementation of Security in Linux

When a user logs in, the login program, *login* (which is SETUID root) asks for a login name and a password. It hashes the password and then looks in the password file, */etc/passwd*, to see if the hash matches the one there (networked systems work slightly differently). The reason for using hashes is to prevent the password from being stored in unencrypted form anywhere in the system. If the password is correct, the login program looks in */etc/passwd* to see the name of the user's preferred shell, possibly *bash*, but possibly some other shell such as *csh* or *ksh*. The login program then uses setuid and setgid to give itself the user's UID and GID (remember, it started out as SETUID root). Then it opens the keyboard for standard input (file descriptor 0), the screen for standard output (file descriptor 1), and

the screen for standard error (file descriptor 2). Finally, it executes the preferred shell, thus terminating itself.

At this point, the preferred shell is running with the correct UID and GID and standard input, output, and error all set to their default devices. All processes that it forks off (i.e., commands typed by the user) automatically inherit the shell's UID and GID, so they also will have the correct owner and group. All files they create also get these values.

When any process attempts to open a file, the system first checks the protection bits in the file's i-node against the caller's effective UID and effective GID to see if the access is permitted. If so, the file is opened and a file descriptor returned. If not, the file is not opened and −1 is returned. No checks are made on subsequent read or write calls. As a consequence, if the protection mode changes after a file is already open, the new mode will not affect processes that already have the file open.

The Linux security model and its implementation are essentially the same as in most other traditional UNIX systems.

10.8 ANDROID

Android is a relatively new operating system designed to run on mobile devices. It is based on the Linux kernel—Android introduces only a few new concepts to the Linux kernel itself, using most of the Linux facilities you are already familiar with (processes, user IDs, virtual memory, file systems, scheduling, etc.) in sometimes very different ways than they were originally intended.

Since its introduction in 2008, Android has grown to be the most widely used operating systems in the world with, as of this writing, over 3 billion monthly active users of just the Google flavor of Android alone. Its popularity has ridden the explosion of smartphones, and it is freely available for manufacturers of mobile devices to use in their products. It is also an open-source platform, making it customizable to a diverse variety of devices. It is popular not only for consumer-centric devices where its third-party application ecosystem is advantageous (such as tablets, televisions, game systems, and media players), but is increasingly used as the embedded OS for dedicated devices that need a graphical user interface such as smart watches, automotive dashboards, airplane seatbacks, medical devices, and home appliances.

A large amount of the Android operating system is written in a high-level language, the Java programming language. The kernel and a large number of low-level libraries are written in C and C++. However, a large amount of the system is written in Java and, but for some small exceptions, the entire application API is written and published in Java as well. The parts of Android written in Java tend to follow a very object-oriented design as encouraged by that language.

10.8.1 Android and Google

Android is an unusual operating system in the way it combines open-source code with closed-source third-party applications. The open-source part of Android is called the **AOSP** (**Android Open Source Project**) and is completely open and free to be used and modified by anyone.

An important goal of Android is to support a rich third-party application environment, which requires having a stable implementation and API for applications to run against. However, in an open-source world where every device manufacturer can customize the platform however it wants, compatibility issues quickly arise. There needs to be some way to control this conflict.

Part of the solution to this for Android is the **CDD** (**Compatibility Definition Document**), which describes the ways Android must behave to be compatible with third party applications. This document describes what is required to be a compatible Android device. Without some way to enforce such compatibility, however, it will often be ignored; there needs to be some additional mechanism to do this.

Android solves this by allowing additional proprietary services to be created on top of the open-source platform, providing (typically cloud-based) services that the platform cannot itself implement. Since these services are proprietary, they can restrict which devices are allowed to include them, thus requiring CDD compatibility of those devices.

Google implemented Android to be able to support a wide variety of proprietary cloud services, with Google's extensive set of services being representative cases: Gmail, calendar and contacts sync, cloud-to-device messaging, and many other services, some visible to the user, some not. When it comes to offering compatible apps, the most important service is Google Play.

Google Play is Google's online store for Android apps. Generally when developers create Android applications, they will publish with Google Play. Since Google Play (or any other application store) is a significant channel through which applications are delivered to an Android device, that proprietary service is responsible for ensuring that applications will work on the devices it delivers them to.

Google Play uses two main mechanisms to ensure compatibility. The first and most important is requiring that any device shipping with it must be a compatible Android device as per the CDD. This ensures a baseline of behavior across all devices. In addition, Google Play must know about any features of a device that an application requires (such as having a touch screen, camera hardware, or telephony support) so the application is not made available on devices that lack them.

10.8.2 History of Android

Google developed Android in the mid-2000s, after acquiring Android as a startup company early in its development. Nearly all the development of the Android platform that exists today was done under Google's management.

Early Development

Android, Inc. was a software company founded to build software to create smarter mobile devices. Originally looking at cameras, the vision soon switched to smartphones due to their larger potential market. That initial goal grew to addressing the then-current difficulty in developing for mobile devices, by bringing to them an open platform built on top of Linux that could be widely used.

During this time, prototypes for the platform's user interface were implemented to demonstrate the ideas behind it. The platform itself was targeting three key languages, JavaScript, Java, and C++, in order to support a rich application-development environment.

Google acquired Android in July 2005, providing the necessary resources and cloud-service support to continue Android development as a complete product. A fairly small group of engineers worked closely together during this time, starting to develop the core infrastructure for the platform and foundations for higher-level application development.

In early 2006, a significant shift in plan was made: instead of supporting multiple programming languages, the platform would focus entirely on the Java programming language for its application development. This was a difficult change, as the original multilanguage approach superficially kept everyone happy with "the best of all worlds"; focusing on one language felt like a step backward to engineers who preferred other languages.

Trying to make everyone happy, however, can easily make nobody happy. Building out three different sets of language APIs would have required much more effort than focusing on a single language, greatly reducing the quality of each one. The decision to focus on the Java language was critical for the ultimate quality of the platform and the development team's ability to meet important deadlines.

As development progressed, the Android platform was developed closely with the applications that would ultimately ship on top of it. Google already had a wide variety of services—including Gmail, Maps, Calendar, YouTube, and of course Search—that would be delivered on top of Android. Knowledge gained from implementing these applications on top of the early platform was fed back into its design. This iterative process with the applications allowed many design flaws in the platform to be addressed early in its development.

Most of the early application development was done with little of the underlying platform actually available to the developers. The platform was usually running all inside one process, through a "simulator" that ran all of the system and applications as a single process on a host computer. In fact there are still some remnants of this old implementation around today, with things like the Application.onTerminate method still in the **SDK** (**Software Development Kit**), which Android programmers use to write applications.

In June 2006, two hardware devices were selected as software-development targets for planned products. The first, code-named "Sooner," was based on an

existing smartphone with a QWERTY keyboard and screen without touch input. The goal of this device was to get an initial product out as soon as possible, by leveraging existing hardware. The second target device, code-named "Dream," was designed specifically for Android, to run it as fully envisioned. It included a large (for that time) touch screen, slide-out QWERTY keyboard, 3G radio (for faster Web browsing), accelerometer, GPS and compass (to support Google Maps), etc.

As the software schedule came better into focus, it became clear that the two hardware schedules did not make sense. By the time it was possible to release Sooner, that hardware would be well out of date, and the effort put on Sooner was pushing out the more important Dream device. To address this, it was decided to drop Sooner as a target device (though development on that hardware continued for some time until the newer hardware was ready) and focus entirely on Dream.

Android 1.0

The first public availability of the Android platform was a preview SDK released in November 2007. This consisted of a hardware device emulator running a full Android device system image and core applications, API documentation, and a development environment. At this point, the core design and implementation were in place, and in most ways closely resembled the modern Android system architecture we will be discussing. The announcement included video demos of the platform running on top of both the Sooner and Dream hardware.

Early development of Android had been done under a series of quarterly demo milestones to drive and show continued process. The SDK release was the first more formal release for the platform. It required taking all the pieces that had been put together so far for application development, cleaning them up, documenting them, and creating a cohesive development environment for third-party developers.

Development now proceeded along two tracks: taking in feedback about the SDK to further refine and finalize APIs, and finishing and stabilizing the implementation needed to ship the Dream device. A number of public updates to the SDK occurred during this time, culminating in a 0.9 release in August 2008 that contained the nearly final APIs.

The platform itself had been going through rapid development, and in the spring of 2008 the focus was shifting to stabilization so that Dream could ship. Android at this point contained a large amount of code that had never been shipped as a commercial product, all the way from parts of the C library, through the Dalvik (and later ART) interpreter (which runs the apps), system services, and applications.

Android also contained quite a few novel design ideas that had never been done before, and it was not clear how they would pan out. This all needed to come together as a stable product, and the team spent a few nail-biting months wondering if all of this stuff would actually come together and work as intended.

Finally, in August 2008, the software was stable and ready to ship. Builds went to the factory and started being flashed onto devices. In September, Android 1.0 was launched on the Dream device, now called the T-Mobile G1.

Continued Development

After Android's 1.0 release, development continued at a rapid pace. There were about 15 major updates to the platform over the following 5 years, adding a large variety of new features and improvements from the initial 1.0 release.

The original Compatibility Definition Document basically allowed only for compatible devices that were very much like the T-Mobile G1. Over the following years, the range of compatible devices would greatly expand. Key points of this process were as follows:

1. During 2009, Android versions 1.5 through 2.0 introduced a soft keyboard to remove a requirement for a physical keyboard, much more extensive screen support (both size and pixel density) for lower-end QVGA devices and new larger and higher density devices like the WVGA Motorola Droid, and a new "system feature" facility for devices to report what hardware features they support and applications to indicate which hardware features they require. The latter is the key mechanism Google Play uses to determine application compatibility with a specific device.
2. During 2011, Android versions 3.0 through 4.0 introduced new core support in the platform for 10-inch and larger tablets; the core platform now fully supported device screen sizes everywhere from small QVGA phones, through smartphones and larger "phablets," 7-inch tablets and larger tablets to beyond 10 inches.
3. As the platform provided built-in support for more diverse hardware, not only larger screens but also nontouch devices with or without a mouse, many more types of Android devices appeared. This included TV devices such as Google TV, gaming devices, notebooks, cameras, etc.

Significant development work also went into something not as visible: a cleaner separation of Google's proprietary services from the Android open-source platform.

For Android 1.0, a significant amount of work had been put into having a clean third-party application API and an open-source platform with no dependencies on proprietary Google code. However, the implementation of Google's proprietary code was often not yet cleaned up, having dependencies on internal parts of the platform. Often the platform did not even have facilities that Google's proprietary

code needed in order to integrate well with it. A series of projects were soon undertaken to address these issues:

1. In 2009, Android version 2.0 introduced an architecture for third parties to plug their own sync adapters into platform APIs like the contacts database. Google's code for syncing various data moved to this well-defined SDK API.
2. In 2010, Android version 2.2 included work on the internal design and implementation of Google's proprietary code. This "great unbundling" cleanly implemented many core Google services, from delivering cloud-based system software updates to "cloud-to-device messaging" and other background services, so that they could be delivered and updated separately from the platform.
3. In 2012, a new **Google Play services** application was delivered to devices, containing updated and new features for Google's proprietary nonapplication services. This was the outgrowth of the unbundling work in 2010, allowing proprietary APIs such as cloud-to-device messaging and maps to be fully delivered and updated by Google.

Since then, there have been a regular series of Android releases. Below are the major releases, with select highlights of the changes in each release related to the core operating system. A number of these will be covered in more detail later.

1. **Android 4.2 (2012)**: Added support for multi-user separation (allowing different people to share a device in isolated users). SELinux introduced in non-enforcing mode.
2. **Android 4.3 (2013)**: Extended multi-user to enable "restricted users," can create restricted environments for children, kiosk modes, point of sale systems, etc.
3. **Android 4.4 (2013)**: SELinux now enforced across operating systems. Android Runtime (ART) is introduced as a developer preview and will later replace the original Dalvik virtual machine. ART features ahead-of-time compilation and a new concurrent garbage collector to avoid GC stalls that cause missed UI frames.
4. **Android 5.0 (2014)**: Introduced the JobScheduler, which would be the future foundation for applications to schedule almost all of their background work with the system. Extended multi-user to support "profiles" where two users run concurrently under different identities (typically providing a concurrent personal and work profile that are isolated from each other). Introduced document-centric-recents model, where recent tasks can include documents or other sub-sections of an overall application. Added support for 64-bit apps.

5. **Android 6.0 (2015)**: Permission model changed from install-time to runtime, reflecting a shift in focus from security to privacy and the increasing complexity of mobile applications with a growing number of secondary features. Introduced the original "doze mode" to take a stronger hand in what apps can do in the background. Security is about protected the device and the user from harm caused by outsiders whereas privacy is focused on protecting the user's information from snooping. They are quite different and need different approaches.
6. **Android 7.0 (2016)**: Extended "doze mode" to cover most situations when the screen is off. On all battery-powered devices, managed energy usage to avoid draining the battery too fast is crucial to the user experience, so in Android 7.0 there was more attention to it.
7. **Android 8.0 (2017)**: A new abstraction, called Treble, was introduced between the Android system and lower-level hardware touched by the kernel and drivers. Similar to the HAL (Hardware Abstraction Layer) in the Windows kernel, Treble provides a stable interface between the bulk of Android and hardware-specific kernel and drivers. It is structured like a microkernel with Treble drivers running in separate user-space processes and Binder IPC (covered later) used to communicate with them. It also placed strong limits on how applications could run in the background, as well as differentiation between background vs. foreground for location access.
8. **Android 9 (2018)**: Limited the ability of applications to launch into their foreground interface while running in the background. Introduced "adaptive battery," where a machine-learning system helps guide the system in deciding the importance of background work across applications.
9. **Android 10 (2019)**: Provided user control over an app's ability to access location information while in the background. Introduced "scoped storage" to better control data access across applications that are putting data on external storage (such as SD cards).
10. **Android 11 (2020)**: Allowed the user to select "only this once" for permissions that provide access to continuous personal data: location, camera, and microphone.
11. **Android 12 (2021)**: Gave the user control over coarse vs. fine location access. Introduced a "permissions hub" allowing users to see how applications have been accessing their personal data. Limited other cases (using foreground services) where applications could go into a foreground state from the background.

10.8.3 Design Goals

A number of key design goals for the Android platform evolved during its development:

1. Provide a complete open-source platform for mobile devices. The open-source part of Android is a bottom-to-top operating system stack, including a variety of applications, that can ship as a complete product.
2. Strongly support proprietary third-party applications with a robust and stable API. As previously discussed, it is challenging to maintain a platform that is both truly open source and also stable enough for proprietary third-party applications. Android uses a mix of technical solutions (specifying a very well-defined SDK and division between public APIs and internal implementation) and policy requirements (through the CDD) to address this.
3. Allow all third-party applications, including those from Google, to compete on a level playing field. The Android open source code is designed to be neutral as much as possible to the higher-level system features built on top of it, from access to cloud services (such as data sync or cloud-to-device messaging APIs), to libraries (such as Google's mapping library) and rich services like application stores.
4. Provide an application security model in which users do not have to deeply trust third-party applications and do not need to rely on a gatekeeper (like a carrier) to control which applications can be installed on the device in order to protect them. The operating system itself must protect the user from misbehavior of applications, not only buggy applications that can cause it to crash, but more subtle misuse of the device and the user's data on it. The less users need to trust applications or the sources of those applications, the more freedom they have to try out and install them.
5. Support typical mobile user interaction, where the user often spends short amounts of time in many apps. The mobile experience tends to involve brief interactions with applications: glancing at new received email, receiving and sending an SMS message or IM, going to contacts to place a call, etc. The system needs to optimize for these cases with fast app launch and switch times; the goal for Android has generally been 200 msec to cold start a basic application up to the point of showing a full interactive UI.
6. Manage application processes for users, simplifying the user experience around applications so that users do not have to worry about

closing applications when done with them. Mobile devices also tend to run without the swap space that allows operating systems to fail more gracefully when the current set of running applications requires more RAM than is physically available. To address both of these requirements, the system needs to take a more proactive stance about managing application processes and deciding when they should be started and stopped.

7. Encourage applications to interoperate and collaborate in rich and secure ways. Mobile applications are in some ways a return back to shell commands: rather than the increasingly large monolithic design of desktop applications, they are often targeted and more focused for specific needs. To help support this, the operating system should provide new types of facilities for these applications to collaborate together to create a larger whole.

8. Create a full general-purpose operating system. Mobile devices are a new expression of general purpose computing, not something simpler than our traditional desktop operating systems. Android's design should be rich enough that it can grow to be at least as capable as a traditional operating system.

10.8.4 Android Architecture

Android is built on top of the standard Linux kernel, with only a few significant extensions to the kernel itself that will be discussed later. Once in user space, however, its implementation is quite different from a traditional Linux distribution and uses many of the Linux features you already understand, but in very different ways.

As in a traditional Linux system, Android's first user-space process is *init*, which is the root of all other processes. The daemons Android's *init* process starts are different, however, focused more on low-level details (managing file systems and hardware access) rather than higher-level user facilities like scheduling cron jobs. Android also has an additional layer of processes, those running ART (for Android Runtime which implements the Java language environment); these are responsible for executing all parts of the system implemented in Java.

Figure 10-39 illustrates the basic process structure of Android. First is the *init* process, which spawns a number of low-level daemon processes. One of these is *zygote*, which is the root of the higher-level Java language processes.

Android's *init* does not run a shell in the traditional way, since a typical Android device does not have a local console for shell access. Instead, the daemon process *adbd* listens for remote connections (such as over USB) that request shell

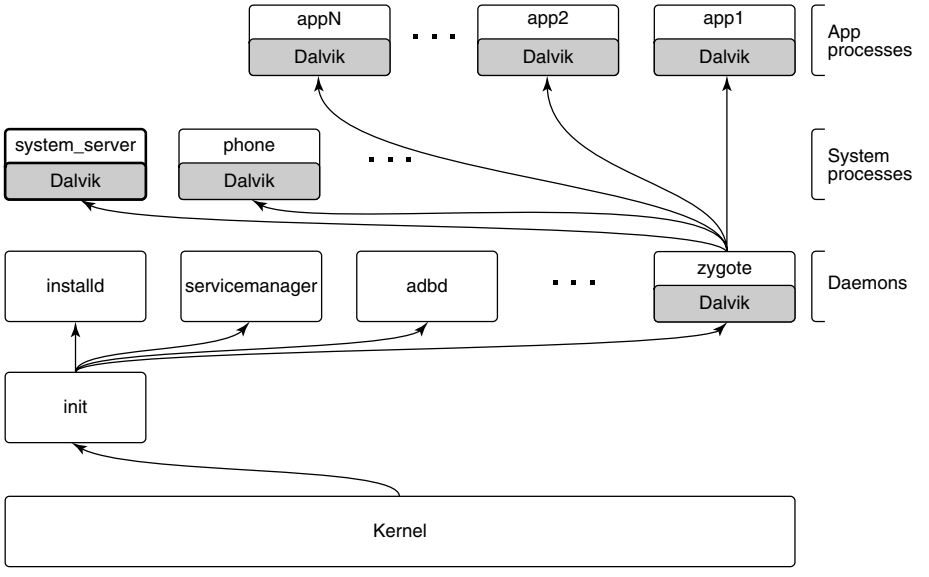

Figure 10-39. Android process hierarchy.

access, forking shell processes for them as needed. These parts are always there, no matter which platform is being used or what features it has.

Since most of Android is written in the Java language, the *zygote* daemon and processes it starts are central to the system. The first process *zygote* always starts is called *system_server*, which contains all of the core operating system services. Key parts of this are the power manager, package manager, window manager, and activity manager.

Other processes will be created from *zygote* as needed. Some of these are "persistent" processes that are part of the basic operating system, such as the telephony stack in the phone process, which must remain always running. Additional application processes will be created and stopped as needed while the system is running.

Applications interact with the operating system through calls to libraries provided by it, which together compose the **Android framework**. Some of these libraries can perform their work within that process, but many will need to perform interprocess communication with other processes, often services in the *system_server* process.

Figure 10-40 shows the typical design for Android framework APIs that interact with system services, in this case the *package manager*. The package manager provides a framework API for applications to call in their local process, here the *PackageManager* class. Internally, this class needs to get a connection to the

coresponding service in the *system_server*. To accomplish this, at boot time the *system_server* publishes each service under a well-defined name in the *service manager*, a daemon started by *init*. The *PackageManager* in the application process retrieves a connection from the *service manager* to its system service using that same name.

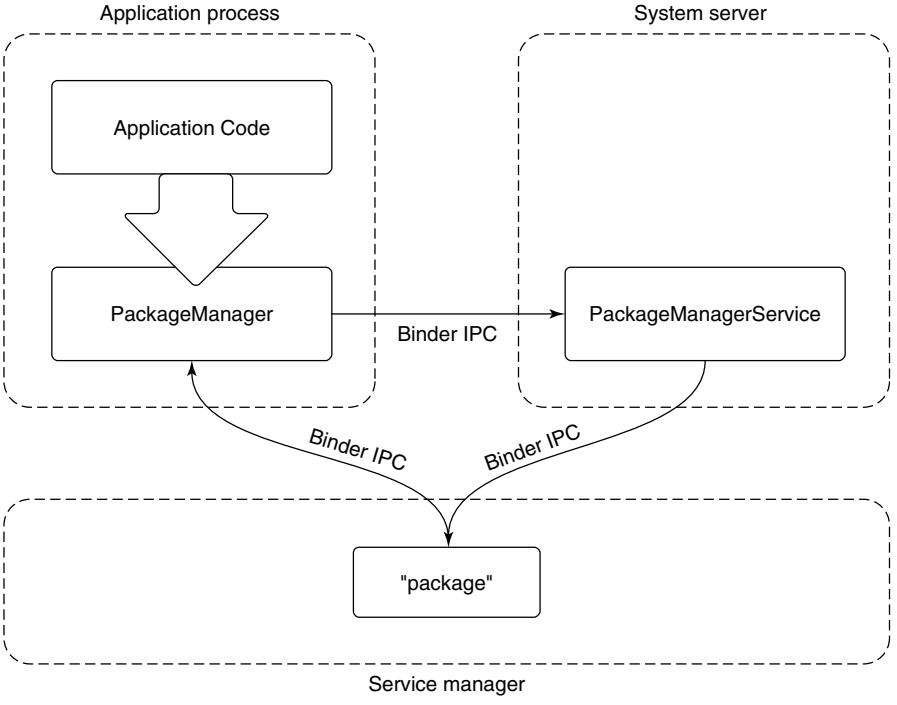

Figure 10-40. Publishing and interacting with system services.

Once the *PackageManager* has connected with its system service, it can make calls on it. Most application calls to *PackageManager* are implemented as interprocess communication using Android's *Binder* IPC mechanism, in this case making calls to the *PackageManagerService* implementation in the *system_server*. The implementation of *PackageManagerService* arbitrates interactions across all client applications and maintains state that will be needed by multiple applications.

10.8.5 Linux Extensions

For the most part, Android includes a stock Linux kernel providing standard Linux features. Most of the interesting aspects of Android as an operating system are in how those existing Linux features are used. There are also, however, several significant extensions to Linux that the Android system relies on.

Wake Locks

Power management on mobile devices is different than on traditional computing systems, so Android adds a new feature to Linux called **wake locks** (also called **suspend blockers**) for managing how the system goes to sleep. This is important in order to save energy and maximize the time before the battery is drained.

On a traditional computing system, the system can be in one of two power states: running and ready for user input, or deeply asleep and unable to continue executing without an external interrupt such as pressing a power key. While running, secondary pieces of hardware may be turned on or off as needed, but the CPU itself and core parts of the hardware must remain in a powered state to handle incoming network traffic and other such events. Going into the lower-power sleep state is something that happens relatively rarely: either through the user explicitly putting the system to sleep, or its going to sleep itself due to a relatively long interval of user inactivity. Coming out of this sleep state requires a hardware interrupt from an external source, such as pressing a key on a keyboard, at which point the device will wake up and turn on its screen.

Mobile device users have different expectations. Although the user can turn off the screen in a way that looks like putting the device to sleep, the traditional sleep state is not actually desired. While a device's screen is off, the device still needs to be able to do work: it needs to be able to receive phone calls, receive and process data for incoming chat messages, and many other things.

The expectations around turning a mobile device's screen on and off are also much more demanding than on a traditional computer. Mobile interaction tends to be in many short bursts throughout the day: you receive a message and turn on the device to see it and perhaps send a one-sentence reply or you run into friends walking their new dog and turn on the device to take a picture of her. In this kind of typical mobile usage, any delay from pulling the device out until it is ready for use has a significant negative impact on the user experience.

Given these requirements, one solution would be to just not have the CPU go to sleep when a device's screen is turned off, so that it is always ready to turn back on again. The kernel does, after all, know when there is no work scheduled for any threads, and Linux (as well as most operating systems) will automatically make the CPU idle and use less power in this situation.

An idle CPU, however, is not the same thing as true sleep. For example:

1. On many chipsets, the idle state uses significantly more power than a true sleep state.
2. An idle CPU can wake up at any moment if some work happens to become available, even if that work is not important.
3. Just having the CPU idle does not tell you that you can turn off other hardware that would not be needed in a true sleep.

Wake locks on Android allow the system to go in to a deeper sleep mode, without being tied to an explicit user action like turning the screen off. The default state of the system with wake locks is that the device is asleep. When the device is running, to keep it from going back to sleep something needs to be holding a wake lock.

While the screen is on, the system always holds a wake lock that prevents the device from going to sleep, so it will stay running, as we expect.

When the screen is off, however, the system itself does not generally hold a wake lock, so it will stay out of sleep only as long as something else is holding one. When no more wake locks are held, the system goes to sleep, and it can come out of sleep only due to a hardware interrupt.

Once the system has gone to sleep, a hardware interrupt will wake it up again, as in a traditional operating system. Some sources of such an interrupt are time-based alarms, events from the cellular radio (such as for an incoming call), incoming network traffic, and presses on certain hardware buttons (such as the power button). Interrupt handlers for these events require one change from standard Linux: they need to acquire an initial wake lock to keep the system running after it handles the interrupt.

The wake lock acquired by an interrupt handler must be held long enough to transfer control up the stack to the driver in the kernel that will continue processing the event. That kernel driver is then responsible for acquiring its own wake lock, after which the interrupt wake lock can be safely released without risk of the system going back to sleep.

If the driver is then going to deliver this event up to user space, a similar handshake is needed. The driver must ensure that it continues to hold the wake lock until it has delivered the event to a waiting user process and ensured there has been an opportunity there to acquire its own wake lock. This flow may continue across subsystems in user space as well; as long as something is holding a wake lock, we continue performing the desired processing to respond to the event. Once no more wake locks are held, however, the entire system falls back to sleep and all processing stops.

After Android shipped, there was significant discussion with the Linux community about how to merge Android's wake lock facility back into the mainline kernel. This was especially important because wake locks require that drivers use them to keep the system running when needed, causing a fork of not just the kernel but also any drivers that need to do this.

Ultimately Linux added a "wakeup event" facility, allowing drivers and other entities in the kernel to note when they are the source of a wakeup and/or need to ensure the device continues to stay way. The decision for whether to go into suspend, however, was moved to user space, keeping the policy for when to suspend out of the kernel. Android provides a user space implementation that makes the decision to suspend based on the wakeup event state in the kernel as well as wake lock requests coming to it from elsewhere in user space.

Out-of-Memory Killer

Linux includes an "out-of-memory killer" that attempts to recover when memory is extremely low. Out-of-memory situations on modern operating systems are nebulous affairs. With paging and swap, it is rare for applications themselves to see out-of-memory failures. However, the kernel can still get in to a situation where it is unable to find available RAM pages when needed, not just for a new allocation, but when swapping in or paging in some address range that is now being used.

In such a low-memory situation, the standard Linux out-of-memory killer is a last resort to try to find RAM so that the kernel can continue with whatever it is doing. This is done by assigning each process a "badness" level, and simply killing the process that is considered the most bad. A process's badness is based on the amount of RAM being used by the process, how long it has been running, and other factors; the goal is to kill large processes that are hopefully not critical.

Android puts special pressure on the out-of-memory killer. It does not have a swap space, so it is much more common to be in out-of-memory situations: there is no way to relieve memory pressure except by dropping clean RAM pages mapped from storage that has been recently used. Even so, Android uses the standard Linux configuration to over-commit memory—that is, allow address space to be allocated in RAM without a guarantee that there is available RAM to back it. Over-commit is an extremely important tool for optimizing memory use, since it is common to mmap large files (such as executables) where you will only be needing to load into RAM small parts of the overall data in that file.

Given this situation, the stock Linux out-of-memory killer does not work well, as it is intended more as a last resort and has a hard time correctly identifying good processes to kill. In fact, as we will discuss later, Android relies extensively on the out-of-memory killer running regularly to reap processes and make good choices about which to select.

To address this, Android introduced its own out-of-memory killer to the kernel, with different semantics and design goals. The Android out-of-memory killer runs much more aggressively: whenever RAM is getting "low." Low RAM is identified by a tunable parameter indicating how much available free and cached RAM in the kernel is acceptable. When the system goes below that limit, the out-of-memory killer runs to release RAM from elsewhere. The goal is to ensure that the system never gets into bad paging states, which can negatively impact the user experience when foreground applications are competing for RAM, since their execution becomes much slower due to continual paging in and out.

Instead of trying to guess which processes are least useful and therefore should be killed, the Android out-of-memory killer relies very strictly on information provided to it by user space. The traditional Linux out-of-memory killer has a per-process *oom_adj* parameter that can be used to guide it toward the best process to kill by modifying the process' overall badness score. Android's original out-of-memory killer used this same parameter, but as a strict ordering: processes with

a higher *oom_adj* will always be killed before those with lower ones. We will discuss later how the Android system decides to assign these scores.

In later versions of Android, a new user-space *lmkd* process was added to take care of killing processes, replacing the original Android implementation in the kernel. This was made possible by newer Linux features such as "pressure-stall information" provided to user space. Switching to *lmkd* not only allows Android to use a closer to stock Linux kernel, but also gives it more flexibility in how the higher-level system interacts with the low-memory-killer.

For example, the *oom_adj* parameter in the kernel has a limit range of values, from −16 to 15. This greatly limits the granularity of process selection that can be provided to it. The new *lmkd* implementation allows a full integer for ordering processes.

10.8.6 ART

ART (**Android RunTime**) implements the Java language environment on Android that is responsible for running applications as well as most of its system code. Almost everything in the *system_service* process—from the package manager, through the window manager, to the activity manager—is implemented with Java language code executed by ART.

Android is not, however, a Java-language platform in the traditional sense. Java code in an Android application is provided in ART's bytecode format, called **DEX** (**Dalvik Executable**), based around a register machine rather than Java's traditional stack-based bytecode.

DEX allows for faster interpretation, while still supporting **JIT** (**Just-in-Time**) compilation. DEX is also more space efficient, both on disk and in RAM, through the use of string pooling and other techniques.

When writing Android applications, source code is written in Java and then compiled into standard Java bytecode using traditional Java tools. Android then introduces a new step: converting that Java bytecode into DEX. It is the DEX version of an application that is packaged up as the final application binary and ultimately installed on the device.

Android's system architecture leans heavily on Linux for system primitives, including memory management, security, and communication across security boundaries. It does not use the Java language for core operating system concepts—there is little attempt to abstract away these important aspects of the underlying Linux operating system.

Of particular note is Android's use of processes. Android's design does not rely on the Java language to protect application from each other and the system, but rather takes the traditional operating system approach of process isolation. This means that each application is running in its own Linux process with its own ART environment, as are the *system_server* and other core parts of the platform that are written in Java.

Using processes for this isolation allows Android to leverage all of Linux's features for managing processes, from memory isolation to cleaning up all of the resources associated with a process when it goes away. In addition to processes, instead of using Java's SecurityManager architecture, Android relies exclusively on Linux's security features.

The use of Linux processes and security greatly simplifies the ART environment, since it is no longer responsible for these critical aspects of system stability and robustness. Not incidentally, it also allows applications to freely use native code in their implementation, which is especially important for games which are usually built with C++-based engines.

Mixing processes and the Java language like this does introduce some challenges. Bringing up a fresh Java-language environment can take more than a second, even on modern mobile hardware. Recall one of the design goals of Android, to be able to quickly launch applications, with a target of 200 msec. Requiring that a fresh ART process be brought up for this new application would be well beyond that budget. A 200-msec launch is hard to achieve on mobile hardware, even without needing to initialize a new Java-language environment.

The solution to this problem is the *zygote* native daemon that we briefly mentioned earlier in the chapter. *Zygote* is responsible for bringing up and initializing ART, to the point where it is ready to start running system or application code written in Java. All new ART-based processes (system or application) are forked from *zygote*, allowing them to start execution with the environment already ready to go. This greatly speeds up launching apps.

It is not just ART that *zygote* brings up. *Zygote* also preloads many parts of the Android framework that are commonly used in the system and application, as well as loading resources and other things that are often needed.

Note that creating a new process from *zygote* involves a Linux fork system call but there is no exec system call. The new process is a replica of the original *zygote* process, with all of its preinitialized state already set up and ready to go. Figure 10-41 illustrates how a new Java application process is related to the original *zygote* process. After the fork, the new process has its own separate ART environment, though it is sharing all of the preloaded and initialed data with *zygote* through copy-on-write pages. All that now needs to be done to have the new running process ready to go is to give it the correct identity (UID, etc.), finish any initialization of ART that requires starting threads, and loading the application or system code to be run.

In addition to launch speed, there is another benefit that *zygote* brings. Because only a fork is used to create processes from it, the large number of dirty RAM pages needed to initialize ART and preload classes and resources can be shared between *zygote* and all of its child processes. This sharing is especially important for Android's environment, where swap is not available; demand paging of clean pages (such as executable code) from "disk" (flash memory) is available. However, any dirty pages must stay locked in RAM; they cannot be paged out to "disk."

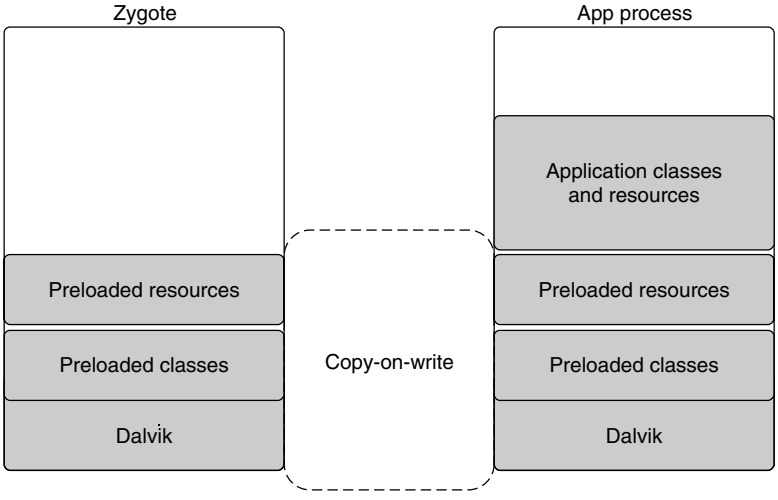

Figure 10-41. Creating a new *ART* process from *zygote*.

10.8.7 Binder IPC

Android's system design revolves significantly around process isolation, between applications as well as between different parts of the system itself. This requires a large amount of interprocess communication to coordinate between the different processes, which can take a large amount of work to implement and get right. Android's *Binder* interprocess communication mechanism is a rich general-purpose IPC facility that most of the Android system is built on top of.

The *Binder* architecture is divided into three layers, shown in Fig. 10-42. At the bottom of the stack is a kernel module that implements the actual cross-process interaction and exposes it through the kernel's *ioctl* function. (*ioctl* is a general-purpose kernel call for sending custom commands to kernel drivers and modules.) On top of the kernel module is a basic object-oriented user-space API, allowing applications to create and interact with IPC endpoints through the *IBinder* and *Binder* classes. At the top is an interface-based programming model where applications declare their IPC interfaces and do not otherwise need to worry about the details of how IPC happens in the lower layers.

Binder Kernel Module

Rather than use existing Linux IPC facilities such as pipes, *Binder* includes a special kernel module that implements its own IPC mechanism. The *Binder* IPC model is different enough from traditional Linux mechanisms that it cannot be efficiently implemented on top of them purely in user space. In addition, Android

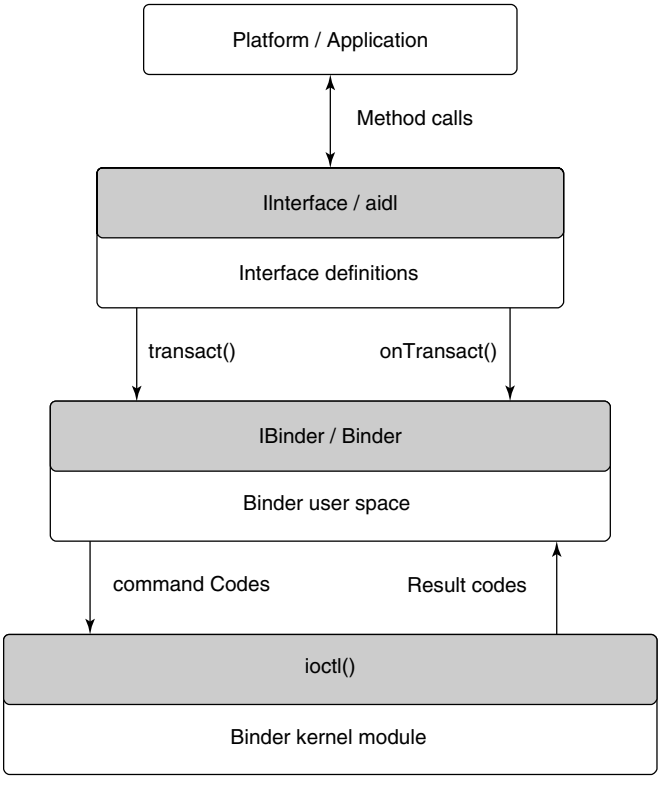

Figure 10-42. *Binder* IPC architecture.

does not support most of the System V primitives for cross-process interaction (semaphores, shared memory segments, message queues) because they do not provide robust semantics for cleaning up their resources from buggy or malicious applications.

The basic IPC model *Binder* uses is the **RPC** (**Remote Procedure Call**). That is, the sending process is submitting a complete IPC operation to the kernel, which is executed in the receiving process; the sender may block while the receiver executes, allowing a result to be returned back from the call. (Senders optionally may specify they should not block, continuing their execution in parallel with the receiver.) *Binder* IPC is thus message based, like System V message queues, rather than stream based as in Linux pipes. A message in *Binder* is referred to as a **transaction**, and at a higher level can be viewed as a function call across processes.

Each transaction that user space submits to the kernel is a complete operation: it identifies the target of the operation and identity of the sender as well as the

complete data being delivered. The kernel determines the appropriate process to receive that transaction, delivering it to a waiting thread in the process.

Figure 10-43 illustrates the basic flow of a transaction. Any thread in the originating process may create a transaction identifying its target, and submit this to the kernel. The kernel makes a copy of the transaction, adding to it the identity of the sender. It determines which process is responsible for the target of the transaction and wakes up a thread in the process to receive it. Once the receiving process is executing, it determines the appropriate target of the transaction and delivers it.

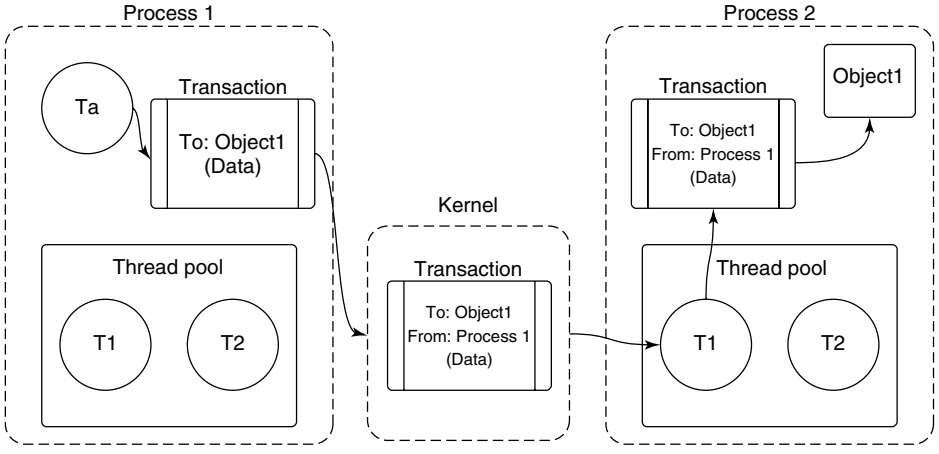

Figure 10-43. Basic *Binder* IPC transaction.

(For the discussion here, we are simplifying the way transaction data moves through the system as two copies, one to the kernel and one to the receiving process's address space. The actual implementation does this in one copy. For each process that can receive transactions, the kernel creates a shared memory area with it. When it is handling a transaction, it first determines the process that will be receiving that transaction and copies the data directly into that shared address space.)

Note that each process in Fig. 10-43 has a "thread pool." This is one or more threads created by user space to handle incoming transactions. The kernel will dispatch each incoming transaction to a thread currently waiting for work in that process's thread pool. Calls into the kernel from a sending process, however, do not need to come from the thread pool—any thread in the process is free to initiate a transaction, such as *Ta* in Fig. 10-43.

We have already seen that transactions given to the kernel identify a target *object*; however, the kernel must determine the receiving *process*. To accomplish this, the kernel keeps track of the available objects in each process and maps them

to other processes, as shown in Fig. 10-44. The objects we are looking at here are simply locations in the address space of that process. The kernel only keeps track of these object addresses, with no meaning attached to them; they may be the location of a C data structure, C++ object, or anything else located in that process's address space.

References to objects in remote processes are identified by an integer *handle*, which is much like a Linux file descriptor. For example, consider *Object2a* in *Process 2*—this is known by the kernel to be associated with *Process 2*, and further the kernel has assigned *Handle 2* for it in *Process 1*. *Process 1* can thus submit a transaction to the kernel targeted to its *Handle 2*, and from that the kernel can determine this is being sent to *Process 2* and specifically *Object2a* in that process.

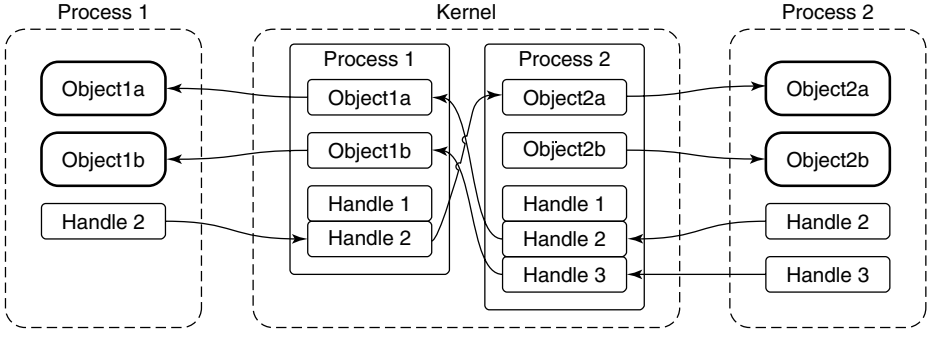

Figure 10-44. *Binder* cross-process object mapping.

Also like file descriptors, the value of a handle in one process does not mean the same thing as that value in another process. For example, in Fig. 10-44, we can see that in *Process 1*, a handle value of 2 identifies *Object2a*; however, in *Process 2*, that same handle value of 2 identifies *Object1a*. Further, it is impossible for one process to access an object in another process if the kernel has not assigned a handle to it for *that process*. Again in Fig. 10-44, we can see that *Process 2*'s *Object2b* is known by the kernel, but no handle has been assigned to it for *Process 1*. There is thus no path for *Process 1* to access that object, even if the kernel has assigned handles to it for other processes.

How do these handle-to-object associations get set up in the first place? Unlike Linux file descriptors, user processes do not directly ask for handles. Instead, the kernel assigns handles to processes as needed. This process is illustrated in Fig. 10-45. Here we are looking at how the reference to *Object1b* from *Process 2* to *Process 1* in the previous figure may have come about. The key to this is how a transaction flows through the system, from left to right at the bottom of the figure.

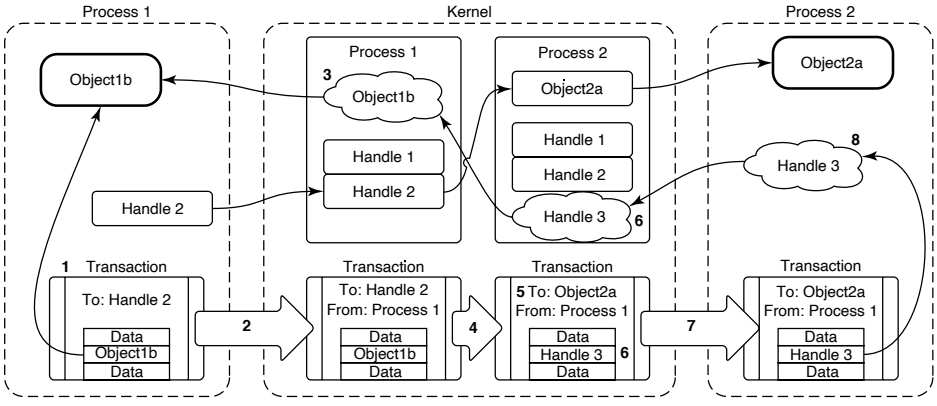

Figure 10-45. Transferring *Binder* objects between processes.

The key steps shown in Fig. 10-45 are as follows:

1. *Process 1* creates the initial transaction structure, which contains the local address *Object1b*.
2. *Process 1* submits the transaction to the kernel.
3. The kernel looks at the data in the transaction, finds the address *Object1b*, and creates a new entry for it since it did not previously know about this address.
4. The kernel uses the target of the transaction, *Handle 2*, to determine that this is intended for *Object2a* which is in *Process 2*.
5. The kernel now rewrites the transaction header to be appropriate for *Process 2*, changing its target to address *Object2a*.
6. The kernel likewise rewrites the transaction data for the target process; here it finds that *Object1b* is not yet known by *Process 2*, so a new *Handle 3* is created for it.
7. The rewritten transaction is delivered to *Process 2* for execution.
8. Upon receiving the transaction, the process discovers there is a new *Handle 3* and adds this to its table of available handles.

If an object within a transaction is already known to the receiving process, the flow is similar, except that now the kernel only needs to rewrite the transaction so that it contains the previously assigned handle or the receiving process's local object pointer. This means that sending the same object to a process multiple times will always result in the same identity, unlike Linux file descriptors where opening

the same file multiple times will allocate a different descriptor each time. The *Binder* IPC system maintains unique object identities as those objects move between processes.

The *Binder* architecture essentially introduces a capability-based security model to Linux. Each *Binder* object is a capability. Sending an object to another process grants that capability to the process. The receiving process may then make use of whatever features the object provides. A process can send an object out to another process, later receive an object from any process, and identify whether that received object is exactly the same object it originally sent out.

Binder User-Space API

Most user-space code does not directly interact with the *Binder* kernel module. Instead, there is a user-space object-oriented library that provides a simpler API. The first level of these user-space APIs maps fairly directly to the kernel concepts we have covered so far, in the form of three classes:

1. **IBinder** is an abstract interface for a *Binder* object. Its key method is *transact*, which submits a transaction to the object. The implementation receiving the transaction may be an object either in the local process or in another process; if it is in another process, this will be delivered to it through the *Binder* kernel module as previously discussed.
2. **Binder** is a concrete *Binder* object. Implementing a *Binder* subclass gives you a class that can be called by other processes. Its key method is *onTransact*, which receives a transaction that was sent to it. The main responsibility of a *Binder* subclass is to look at the transaction data it receives here and perform the appropriate operation.
3. **Parcel** is a container for reading and writing data that are in a *Binder* transaction. It has methods for reading and writing typed data—integers, strings, arrays—but most importantly it can read and write references to any *IBinder* object, using the appropriate data structure for the kernel to understand and transport that reference across processes.

Figure 10-46 depicts how these classes work together, modifying Fig. 10-44 that we previously looked at with the user-space classes that are used. Here we see that *Binder1b* and *Binder2a* are instances of concrete *Binder* subclasses. To perform an IPC, a process now creates a **Parcel** containing the desired data, and sends it through another class we have not yet seen, **BinderProxy**. This class is created whenever a new handle appears in a process, thus providing an implementation of *IBinder* whose *transact* method creates the appropriate transaction for the call and submits it to the kernel.

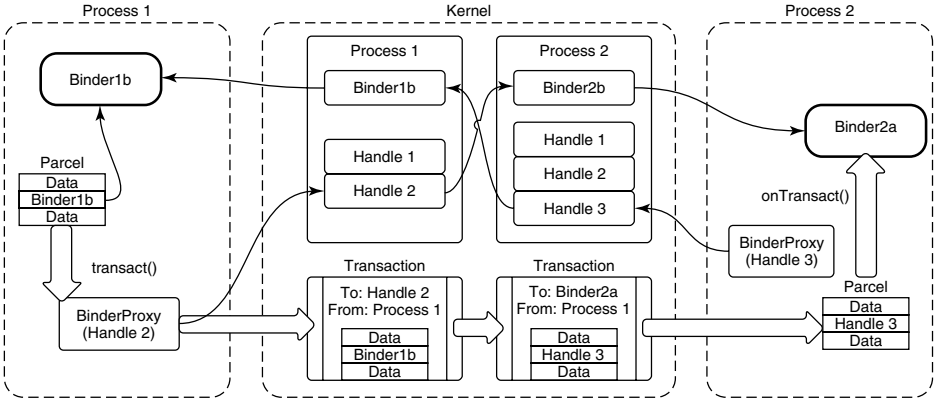

Figure 10-46. *Binder* user-space API.

The kernel transaction structure we had previously looked at is thus split apart in the user-space APIs: the target is represented by a *BinderProxy* and its data are held in a *Parcel*. The transaction flows through the kernel as we previously saw and, upon appearing in user space in the receiving process, its target is used to determine the appropriate receiving *Binder* object while a *Parcel* is constructed from its data and delivered to that object's *onTransact* method.

These three classes now make it fairly easy to write IPC code:

1. Subclass from *Binder*.
2. Implement *onTransact* to decode and execute incoming calls.
3. Implement corresponding code to create a *Parcel* that can be passed to that object's *transact* method.

The bulk of this work is in the last two steps. This is the **unmarshalling** and **marshalling** code that is needed to turn how we'd prefer to program—using simple method calls—into the operations that are needed to execute an IPC. This is boring and error-prone code to write, so we'd like to let the computer take care of that for us.

Binder Interfaces and AIDL

The final piece of *Binder* IPC is the one that is most often used, a high-level interface-based programming model. Instead of dealing with *Binder* objects and **Parcel** data, here we get to think in terms of interfaces and methods.

The main piece of this layer is a command-line tool called **AIDL** (for **Android Interface Definition Language**). This tool is an interface compiler, taking an abstract description of an interface and generating from it the source code that is

necessary to define that interface and implement the appropriate marshalling and unmarshalling code needed to make remote calls with it.

Figure 10-47 shows a simple example of an interface defined in AIDL. This interface is called *IExample* and contains a single method, *print*, which takes a single String argument.

```
package com.example
interface IExample {
     void print(String msg);
}
```

Figure 10-47. Simple interface described in AIDL.

An interface description like that in Fig. 10-47 is compiled by AIDL to generate three Java-language classes illustrated in Fig. 10-48:

1. **IExample** supplies the Java-language interface definition.
2. **IExample.Stub** is the base class for implementations of this interface. It inherits from *Binder*, meaning it can be the recipient of IPC calls; it inherits from **IExample**, since this is the interface being implemented. The purpose of this class is to perform unmarshalling: turn incoming *onTransact* calls in to the appropriate method call of *IExample*. A subclass of it is then responsible only for implementing the *IExample* methods.
3. **IExample.Proxy** is the other side of an IPC call, responsible for performing marshalling of the call. It is a concrete implementation of *IExample*, implementing each method of it to transform the call into the appropriate **Parcel** contents and send it off through a *transact* call on an *IBinder* it is communicating with.

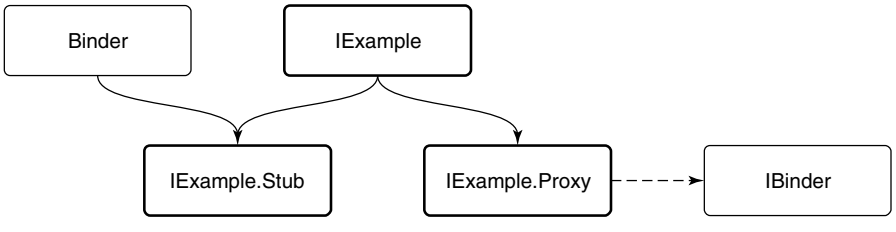

Figure 10-48. *Binder* interface inheritance hierarchy.

With these classes in place, there is no longer any need to worry about the mechanics of an IPC. Implementors of the *IExample* interface simply derive from *IExample.Stub* and implement the interface methods as they normally would.

Callers will receive an *IExample* interface that is implemented by *IExample.Proxy*, allowing them to make regular calls on the interface.

The way these pieces work together to perform a complete IPC operation is shown in Fig. 10-49. A simple *print* call on an *IExample* interface turns into:

1. *IExample.Proxy* marshals the method call into a *Parcel*, calling *transact* on the *IBinder* it is connected to, which is typically a *BinderProxy* for an object in another process.
2. *BinderProxy* constructs a kernel transaction and delivers it to the kernel through an *ioctl* call.
3. The kernel transfers the transaction to the intended process, delivering it to a thread that is waiting in its own *ioctl* call.
4. The transaction is decoded back into a *Parcel* and *onTransact* called on the appropriate local object, here *ExampleImpl* (which is a subclass of *IExample.Stub*).
5. *IExample.Stub* decodes the *Parcel* into the appropriate method and arguments to call, here calling *print*.
6. The concrete implementation of *print* in *ExampleImpl* finally executes.

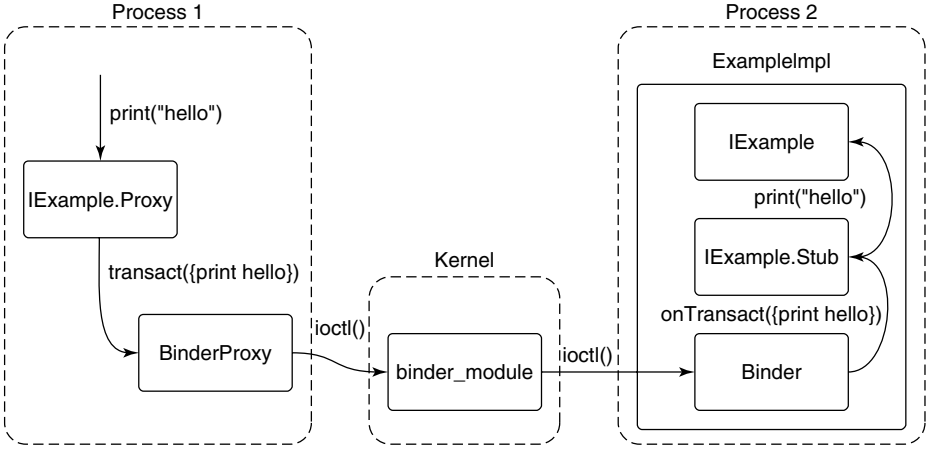

Figure 10-49. Full path of an *AIDL*-based *Binder* IPC.

The bulk of Android's IPC is written using this mechanism. Most services in Android are defined through AIDL and implemented as shown here. Recall the previous Fig. 10-40 showing how the implementation of the *package manager* in the *system_server* process uses IPC to publish itself with the *service manager* for

other processes to make calls to it. Two AIDL interfaces are involved here: one for the *service manager* and one for the *package manager*. For example, Fig. 10-50 shows the basic AIDL description for the *service manager*; it contains the *getService* method, which other processes use to retrieve the *IBinder* of system service interfaces like the *package manager*.

```
package android.os

interface IServiceManager {
    IBinder getService(String name);
    void addService(String name, IBinder binder);
}
```

Figure 10-50. Basic service manager AIDL interface.

10.8.8 Android Applications

Android provides an application model that is very different from a typical command-line environment in the Linux shell or even applications launched from a graphical user interface such as Gnome or KDE. An application is *not* an executable file with a main entry point; it is a container of everything that makes up that app: its code, graphical resources, declarations about what it is to the system, and other data.

An Android application by convention is a file with the *apk* extension, for **Android Package**. This file is actually a normal *zip* archive, containing everything about the application. The important contents of an *apk* are as follows:

1. A manifest describing what the application is, what it does, and how to run it. The manifest must provide a package name for the application, a Java-style scoped string (such as com.android.app.calculator), which uniquely identifies it.
2. Resources needed by the application, including strings it displays to the user, XML data for layouts and other descriptions, graphical bitmaps, etc.
3. The code itself, which may be ART bytecode as well as native library code.
4. Signing information, securely identifying the author.

The key part of the application for our purposes here is its manifest, which appears as a precompiled XML file named AndroidManifest.xml in the root of the apk's zip namespace. A complete example manifest declaration for a hypothetical email application is shown in Fig. 10-51: it allows you to view and compose emails

and also includes components needed for synchronizing its local email storage with a server even when the user is not currently in the application.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
      package="com.example.email">
   <application>

      <activity android:name="com.example.email.MailMainActivity">
         <intent-filter>
            <action android:name="android.intent.action.MAIN" />
            <category android:name="android.intent.category.LAUNCHER" />
         </intent-filter>
      </activity>

      <activity android:name="com.example.email.ComposeActivity">
         <intent-filter>
            <action android:name="android.intent.action.SEND" />
            <category android:name="android.intent.category.DEFAULT" />
            <data android:mimeType="*/*" />
         </intent-filter>
      </activity>

      <service android:name="com.example.email.SyncService">
      </service>

      <receiver android:name="com.example.email.SyncControlReceiver">
         <intent-filter>
            <action android:name="android.intent.action.DEVICE_STORAGE_LOW" />
         </intent-filter>
         <intent-filter>
            <action android:name="android.intent.action.DEVICE_STORAGE_OKAY" />
         </intent-filter>
      </receiver>

      <provider android:name="com.example.email.EmailProvider"
            android:authorities="com.example.email.provider.email">
      </provider>

   </application>
</manifest>
```

Figure 10-51. Basic structure of AndroidManifest.xml.

Keep in mind that while what is described here is a real application you could write for Android, in order to focus on illustrating key operating system concepts the example has been simplified and modified from how an actual application like this is typically designed. If you have written an Android application and seeing this example makes you feel like something is off, you are not wrong!

Android applications do not have a simple main entry point that is executed when the user launches them. Instead, they publish under the manifest's <application> tag a variety of entry points describing the various things the application can do. These entry points are expressed as four distinct types, defining the core types of behavior that applications can provide: activity, receiver, service, and content provider. The example we have presented shows a few activities and one declaration of the other component types, but an application may declare zero or more of any of these.

Each of the different four component types an application can contain has different semantics and uses within the system. In all cases, the android:name attribute supplies the Java class name of the application code implementing that component, which will be instantiated by the system when needed.

The **package manager** is the part of Android that keeps track of all application packages. When a user downloads an app, it comes in a package containing everything the app needs. It parses every application's manifest, collecting and indexing the information it finds in them. With that information, it then provides facilities for clients to query it about the app information those clients are allowed to access, such as whether an app is currently installed and the kinds of things an app can do. It is also responsible for installing applications (creating storage space for the application and ensuring the integrity of the apk) as well as everything needed to uninstall an app, which includes cleaning up everything associated with a previously installed version of the app.

Applications statically declare their entry points in their manifest so they do not need to execute code at install time that registers them with the system. This design makes the system more robust in many ways: since installing an application does not run any application code and the top-level capabilities of the application can always be determined by looking at its manifest, there is no need to keep a separate database of this information about the application which can get out of sync (such as across updates) with the application's actual capabilities, and it guarantees no information about an application can be left around after it is uninstalled. This decentralized approach was taken to avoid many of these types of problems caused by Windows' centralized Registry.

Breaking an application into finer-grained components also serves our goal of supporting interoperation and collaboration between applications. Applications can publish pieces of themselves that provide specific functionality, which other applications can make use of either directly or indirectly. This will be illustrated as we look in more detail at the four kinds of components that can be published.

Above the package manager sits another important system service, the **activity manager**. While the package manager is responsible for maintaining static information about all installed applications, the activity manager determines when, where, and how those applications should run. Despite its name, it is actually responsible for running all four types of application components and implementing the appropriate behavior for each of them.

Activities

An **activity** is a part of the application that interacts directly with the user through a user interface. When the user launches an application on their device, this is actually an activity inside the application that has been designated as such a main entry point. The application implements code in its activity that is responsible for interacting with the user.

The example email manifest shown in Fig. 10-51 contains two activities. The first is the main mail user interface, allowing users to view their messages; the second is a separate interface for composing a new message. The first mail activity is declared as the main entry point for the application; that is, the activity that will be started when the user launches it from the home screen.

Since the first activity is the main activity, it will be shown to users as an application they can launch from the main application launcher. If they do so, the system will be in the state shown in Fig. 10-52. Here the activity manager, on the left side, has made an internal *ActivityRecord* instance in its process to keep track of the activity. One or more of these activities are organized into containers called *tasks*, which roughly correspond to what the user experiences as an application. At this point the activity manager has started the email application's process and an instance of its *MainMailActivity* for displaying its main UI, which is associated with the appropriate *ActivityRecord*. This activity is in a state called *resumed* since it is now in the foreground of the user interface.

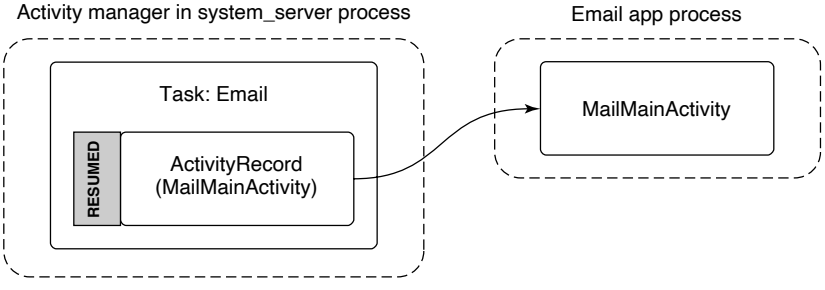

Figure 10-52. Starting an email application's main activity.

If the user were now to switch away from the email application (not exiting it) and launch a camera application to take a picture, we would be in the state shown in Fig. 10-53. Note that we now have a new camera process running the camera's main activity, an associated *ActivityRecord* for it in the activity manager, and it is now the resumed activity. Something interesting also happens to the previous email activity: instead of being resumed, it is now *stopped* and the *ActivityRecord* holds this activity's *saved state*.

When an activity is no longer in the foreground, the system automatically asks it to "save its state." This involves the application creating a minimal amount of

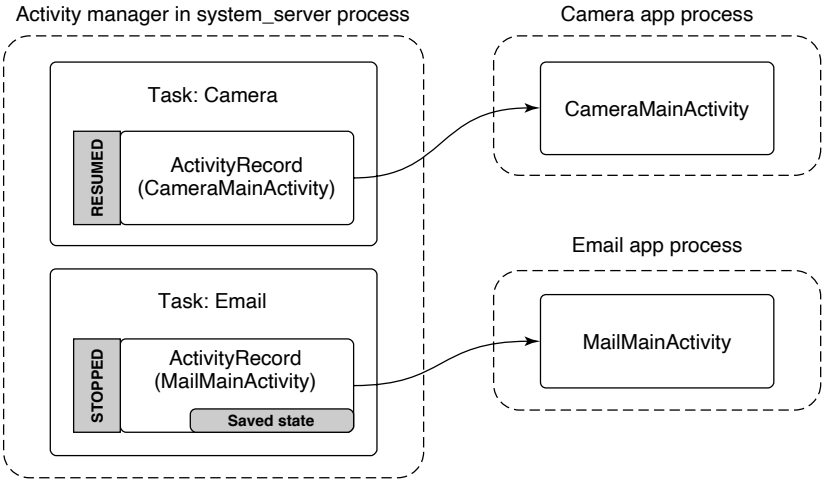

Figure 10-53. Starting the camera application after email.

state information representing what the user currently sees that it returns to the activity manager; the activity manager, running in the *system_server* process, retains that state in its *ActivityRecord* for that activity. The saved state for an activity is generally small, for example containing where you are scrolled in an email message; it would not contain data like the message itself, which the app would instead keep somewhere in its own persistent storage (so it remains around even if the user completely removes an activity).

Recall that although Android does demand paging (it can page in and out clean RAM that has been mapped from files on disk, such as code), it does not rely on swap space. This means all dirty RAM pages in an application's process *must* stay in RAM. Having the email's main activity state safely stored away in the activity manager gives the system back some of the flexibility in dealing with memory that swap provides.

For example, if the camera application starts to require a lot of RAM, the system can simply get rid of the email process, as shown in Fig. 10-54. The *ActivityRecord*, with its precious saved state, remains safely tucked away by the activity manager in the *system_server* process. Since the *system_server* process hosts all of Android's core system services, it must always remain running, so the state saved here will remain around for as long as we might need it.

Our example email application not only has an activity for its main UI, but includes another *ComposeActivity*. Applications can declare any number of activities they want. This can help organize the implementation of an application, but more importantly it can be used to implement cross-application interactions. For example, this is the basis of Android's cross-application sharing system, which the

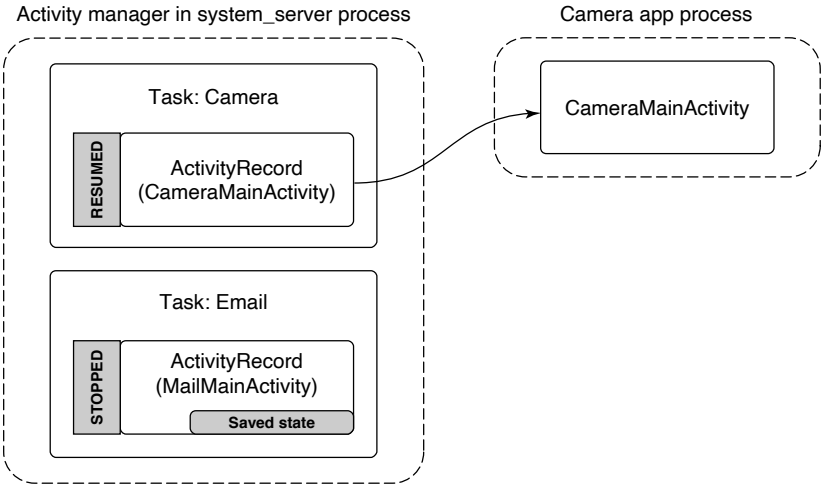

Figure 10-54. Removing the email process to reclaim RAM for the camera.

ComposeActivity here is participating in. If the user, while in the camera application, decides she wants to share a picture she took, our email application's *ComposeActivity* is one of the sharing options she has. If it is selected, that activity will be started and given the picture to be shared. (Later we will see how the camera application is able to find the email application's *ComposeActivity*.)

Performing that share option while in the activity state seen in Fig. 10-54 will lead to the new state in Fig. 10-55. There are a number of important things to note:

1. The email app's process must be started again, to run its *ComposeActivity*.
2. However, the old *MailMainActivity* is *not* started at this point, since it is not needed. This reduces RAM use.
3. The camera's task now has two records: the original *CameraMainActivity* we had just been in, and the new *ComposeActivity* that is now displayed. To the user, these are still one cohesive task: it is the camera currently interacting with them to email a picture.
4. The new *ComposeActivity* is at the top, so it is resumed; the previous *CameraMainActivity* is no longer at the top, so its state has been saved. We can at this point safely quit its process if its RAM is needed elsewhere.

If you want to experiment yourself with this on Android, it should be noted that starting in Android 5.0 a real share flow would result in the *ComposeActivity*

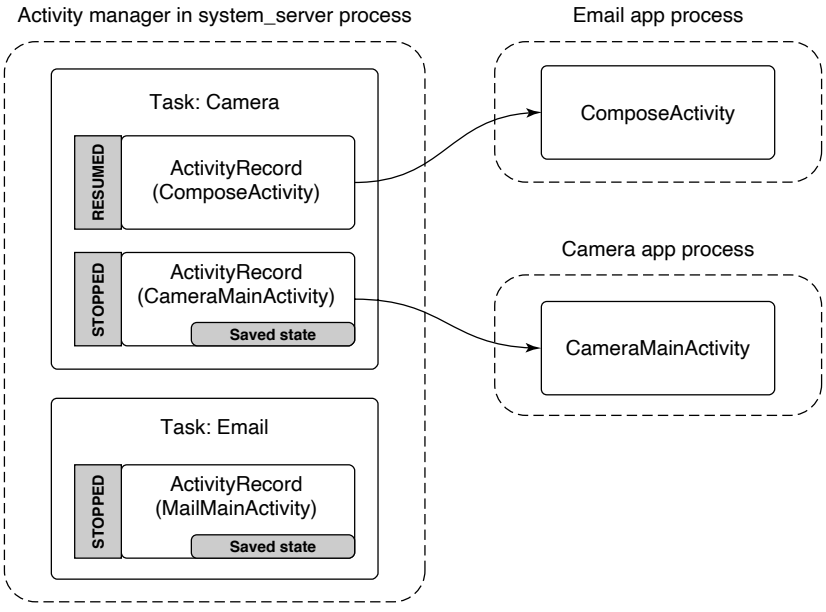

Figure 10-55. Sharing a camera picture through the email application.

appearing in its own third task, separate from *CameraMainActivity*. This was part of a switch to a "document-centric recents" model, described in

https://developer.android.com/guide/components/activities/recents

where the tasks we have here that are shown to users could be contextual parts of apps as well as the apps themselves. The activity abstraction between apps and the operating system allowed implementing this kind of significant user experience with little to no modification of the apps themselves.

Finally, let us look at what would happen if the user left the camera task while in this last state (that is, composing an email to share a picture) and returned to the email application. Figure 10-56 shows the new state the system will be in. Note that we have brought the email task with its main activity back to the foreground. This makes *MailMainActivity* the foreground activity, but there is currently no instance of it running in the application's process.

To return to the previous activity, the system makes a new instance, handing it back the previously saved state the old instance had provided. This action of *restoring an activity from its saved state* must be able to bring the activity back to the same visual state as the user last left it. To accomplish this, the application will look in its saved state for the message the user was in, load that message's data from its persistent storage, and then apply any scroll position or other user-interface state that had been saved.

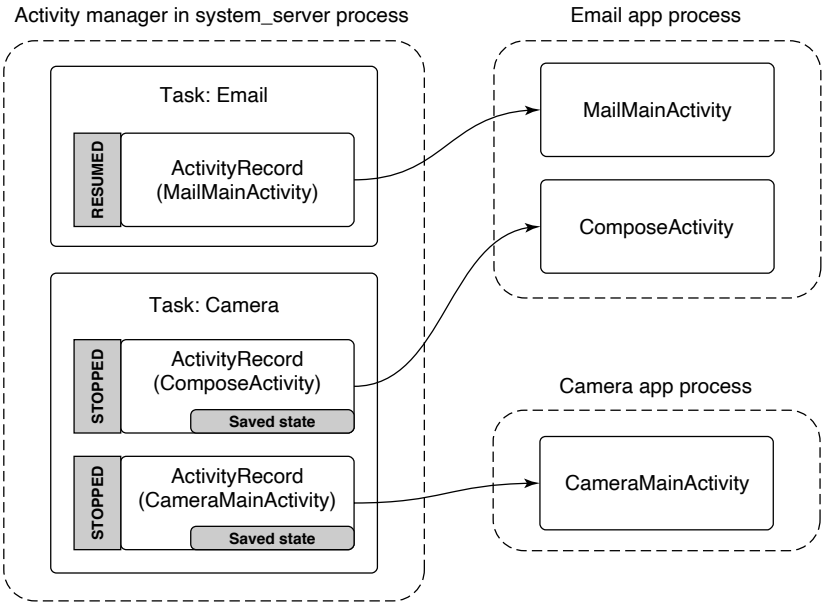

Figure 10-56. Returning to the email application.

Services

A **service** has two distinct identities:

1. It can be a self-contained long-running background operation. Common examples of using services in this way are performing background music playback, maintaining an active network connection (such as with an IRC server) while the user is in other applications, downloading or uploading data in the background, etc.
2. It can serve as a connection point for other applications or the system to perform rich interaction with the application. This can be used by applications to provide secure APIs for other applications, such as to perform image or audio processing, provide a text to speech, etc.

The example email manifest shown in Fig. 10-51 contains a service that is used to perform synchronization of the user's mailbox. A common implementation would schedule the service to run at a regular interval, such as every 15 minutes, *starting* the service when it is time to run, and *stopping* itself when done.

This is a typical use of the first style of service, a long-running background operation. Figure 10-57 shows the state of the system in this case, which is quite simple. The activity manager has created a *ServiceRecord* to keep track of the

service, noting that it has been *started*, and thus created its *SyncService* instance in the application's process. While in this state the service is fully active (barring the entire system going to sleep if not holding a wake lock) and free to do what it wants. It is possible for the application's process to go away while in this state, such as if the process crashes, but the activity manager will continue to maintain its *ServiceRecord* and can at that point decide to restart the service if desired.

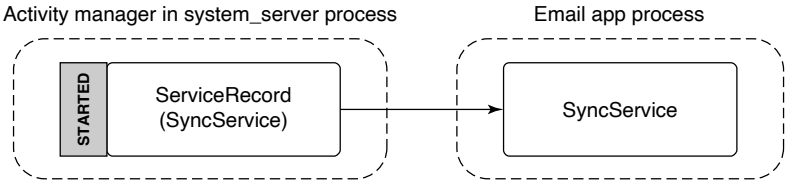

Figure 10-57. Starting an application service.

To see how one can use a service as a connection point for interaction with other applications, let us say that we want to extend our existing *SyncService* to have an API that allows other applications to control its sync interval. We will need to define an AIDL interface for this API, like the one shown in Fig. 10-58.

```
package com.example.email

interface ISyncControl {
      int getSyncInterval();
      void setSyncInterval(int seconds);
}
```

Figure 10-58. Interface for controlling a sync service's sync interval.

To use this, another process can *bind* to our application service, getting access to its interface. This creates a connection between the two applications, shown in Fig. 10-59. The steps of this process are as follows:

1. The client application tells the activity manager that it would like to bind to the service.
2. If the service is not already created, the activity manager creates it in the service application's process.
3. The service returns the *IBinder* for its interface back to the activity manager, which now holds that *IBinder* in its *ServiceRecord*.
4. Now that the activity manager has the service *IBinder*, it can be sent back to the original client application.
5. The client application now having the service's *IBinder* may proceed to make any direct calls it would like on its interface.

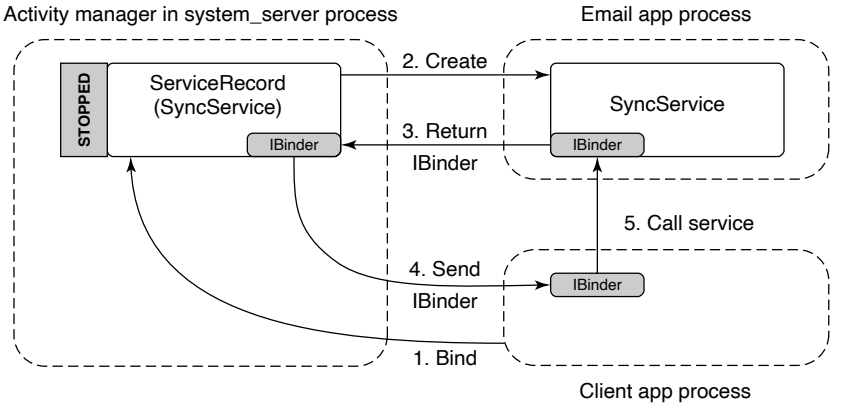

Figure 10-59. Binding to an application service.

Receivers

A **receiver** is the recipient of (typically external) event s that happen, most of the time in the background and outside of normal user interaction with an app. Receivers conceptually are the same as an application explicitly registering for a callback when something interesting happens (an alarm goes off, data connectivity changes, etc.), but do not require that the application be running in order to receive the event.

The example email manifest shown in Fig. 10-51 contains a receiver for the application to find out when the device's storage becomes low in order for it to stop synchronizing email (which may consume more storage). When the device's storage becomes low, the system will send a *broadcast* with the low storage code, to be delivered to all receivers interested in the event.

Figure 10-60 illustrates how such a broadcast is processed by the activity manager in order to deliver it to interested receivers. It first asks the package manager for a list of all receivers interested in the event, which is placed in a *BroadcastRecord* representing that broadcast. The activity manager will then proceed to step through each entry in the list, having each associated application's process create and execute the appropriate receiver class.

Receivers only run as one-shot operations. They are activated only one time. When an event happens, the system finds any receivers interested in it, delivers that event to them, and once they have consumed the event they are done. There is no *ReceiverRecord* like those we have seen for other application components, because a particular receiver is only a transient entity for the duration of a single broadcast. Each time a new broadcast is sent to a receiver component, a new instance of that receiver's class is created.

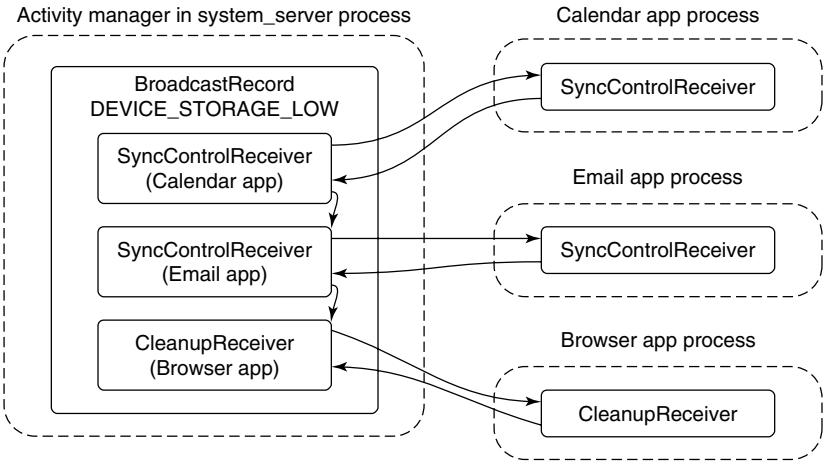

Figure 10-60. Sending a broadcast to application receivers.

Content Providers

Our last application component, the **content provider**, is the primary mechanism that applications use to exchange data with each other. All interactions with a content provider are through URIs using a *content:* scheme; the authority of the URI is used to find the correct content-provider implementation to interact with.

For example, in our email application from Fig. 10-51, the content provider specifies that its authority is *com.example.email.provider.email*. Thus, URIs operating on this content provider would start with

```
content://com.example.email.provider.email/
```

The suffix to that URI is interpreted by the provider itself to determine what data within it is being accessed. In the example here, a common convention would be that the URI

```
content://com.example.email.provider.email/messages
```

means the list of all email messages, while

```
content://com.example.email.provider.email/messages/1
```

provides access to a single message at key number 1.

To interact with a content provider, applications always go through a system API called *ContentResolver*, where most methods have an initial URI argument indicating the data to operate on. One of the most often used *ContentResolver* methods is *query*, which performs a database query on a given URI and returns a

Cursor for retrieving the structured results. For example, retrieving a summary of all of the available email messages would look something like:

```
query("content://com.example.email.provider.email/messages")
```

Though this does not look like it to applications, what is actually going on when they use content providers has many similarities to binding to services. Figure 10-61 illustrates how the system handles our query example:

1. The application calls *ContentResolver.query* to initiate the operation.
2. The URI's authority is handed to the activity manager for it to find (via the package manager) the appropriate content provider.
3. If the content provider is not already running, it is created.
4. Once created, the content provider returns to the activity manager its *IBinder* implementing the system's *IContentProvider* interface.
5. The content provider's *Binder* is returned to the *ContentResolver*.
6. The content resolver can now complete the initial *query* operation by calling the appropriate method on the AIDL interface, returning the *Cursor* result.

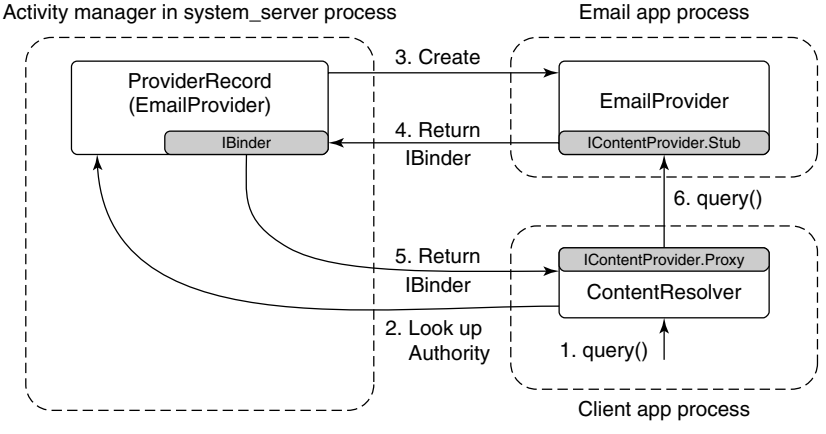

Figure 10-61. Interacting with a content provider.

Content providers are one of the key mechanisms for performing interactions across applications. For example, if we return to the cross-application sharing system previously described in Fig. 10-55, content providers are the way data are actually transferred. The full flow for this operation is:

1. A share request that includes the URI of the data to be shared is created and is submitted to the system.
2. The system asks the *ContentResolver* for the MIME type of the data behind that URI; this works much like the *query* method we just discussed, but asks the content provider to return a MIME-type string for the URI.
3. The system finds all activities that can receive data of the identified MIME type.
4. A user interface is shown for the user to select one of the possible recipients.
5. When one of these activities is selected, the system launches it.
6. The share-handling activity receives the URI of the data to be shared, retrieves its data through *ContentResolver*, and performs its appropriate operation: creates an email, stores it, etc.

10.8.9 Intents

A detail that we have not yet discussed in the application manifest shown in Fig. 10-51 is the <intent-filter> tags included with the activity and receiver declarations. This is part of the **intent** feature in Android, which is the cornerstone for how different applications identify each other in order to be able to interact and work together.

An intent is the mechanism Android uses to discover and identify activities, receivers, and services. It is similar in some ways to the Linux shell's search path, which the shell uses to look through multiple possible directories in order to find an executable matching command names given to it.

There are two major types of intents: *explicit* and *implicit*. An **explicit intent** is one that directly identifies a single specific application component; in Linux shell terms it is the equivalent to supplying an absolute path to a command. The most important part of such an intent is a pair of strings naming the component: the *package name* of the target application and *class name* of the component within that application. Now referring back to the activity of Fig. 10-52 in application Fig. 10-51, an explicit intent for this component would be one with package name com.example.email and class name com.example.email.MailMainActivity.

The package and class name of an explicit intent are enough information to uniquely identify a target component, such as the main email activity in Fig. 10-52. From the package name, the package manager can return everything needed about the application, such as where to find its code. From the class name, we know which part of that code to execute.

An **implicit intent** is one that describes characteristics of the desired component, but not the component itself; in Linux shell terms this is the equivalent to supplying a single command name to the shell, which it uses with its search path to find a concrete command to be run. This process of finding the component matching an implicit intent is called **intent resolution**.

Android's general sharing facility, as we previously saw in Fig. 10-55's illustration of sharing a photo the user took from the camera through the email application, is a good example of implicit intents. Here the camera application builds an intent describing the action to be done, and the system finds all activities that can potentially perform that action. A share is requested through the intent action android.intent.action.SEND, and we can see in Fig. 10-51 that the email application's compose activity declares that it can perform this action.

There can be three outcomes to an intent resolution: (1) no match is found, (2) a single unique match is found, or (3) there are multiple activities that can handle the intent. An empty match will result in either an empty result or an exception, depending on the expectations of the caller at that point. If the match is unique, then the system can immediately proceed to launching the now explicit intent. If the match is not unique, we need to somehow resolve it in another way to a single result.

If the intent resolves to multiple possible activities, we cannot just launch all of them; we need to pick a single one to be launched. This is accomplished through a trick in the package manager. If the package manager is asked to resolve an intent down to a single activity, but it finds there are multiple matches, it instead resolves the intent to a special activity built into the system called the **ResolverActivity**. This activity, when launched, simply takes the original intent, asks the package manager for a list of all matching activities, and displays these for the user to select a single desired action. When one is selected, it creates a new explicit intent from the original intent and the selected activity, calling the system to have that new activity started.

Android has another similarity with the Linux shell: Android's graphical shell, the launcher, runs in user space like any other application. An Android launcher performs calls on the package manager to find the available activities and launch them when selected by the user.

10.8.10 Process Model

The traditional process model in Linux is a fork to create a new process, followed by an exec to initialize that process with the code to be run and then start its execution. The shell is responsible for driving this execution, forking and executing processes as needed to run shell commands. When those commands exit, the process is removed by Linux.

Android uses processes somewhat differently. As discussed in the previous section on applications, the activity manager is the part of Android responsible for

managing running applications. It coordinates the launching of new application processes, determines what will run in them, and when they are no longer needed.

Starting Processes

In order to launch new processes, the activity manager must communicate with the *zygote*. When the activity manager first starts, it creates a dedicated socket with *zygote*, through which it sends a command when it needs to start a process. The command primarily describes the sandbox to be created: the UID that the new process should run (which will be discussed later on security) as and any other security restrictions that will apply to it. *Zygote* thus must run as root: when it forks, it does the appropriate setup for the sandbox it will run in, finally dropping root privileges and changing the process to the desired sandbox.

Recall in our previous discussion about Android applications that the activity manager maintains dynamic information about the execution of activities (in Fig. 10-52), services (Fig. 10-57), broadcasts (to receivers as in Fig. 10-60), and content providers (Fig. 10-61). It uses this information to drive the creation and management of application processes. For example, when the application launcher calls in to the system with a new intent to start an activity as we saw in Fig. 10-52, it is the activity manager that is responsible for making that new application run.

The flow for starting an activity in a new process is shown in Fig. 10-62. The details of each step in the illustration are as follows:

1. Some existing process (such as the app launcher) calls in to the activity manager with an intent describing the new activity it would like to have started.
2. Activity manager asks the package manager to resolve the intent to an explicit component.
3. Activity manager determines that the application's process is not already running, and then asks *zygote* for a new process of the appropriate UID.
4. *Zygote* performs a fork, creating a new process that is a clone of itself, drops privileges and sets up its sandbox appropriately, and finishes initialization of ART in that process so that the Java runtime is fully executing. For example, it must start threads like the garbage collector after it forks.
5. The new process, now a clone of *zygote* with the Java environment fully up and running, calls back to the activity manager, asking "What am I supposed to do?"
6. Activity manager returns back the full information about the application it is starting, such as where to find its code.

7. New process loads the code for the application being run.
8. Activity manager sends to the new process any pending operations, in this case "start activity X."
9. New process receives the command to start an activity, instantiates the appropriate Java class, and executes it.

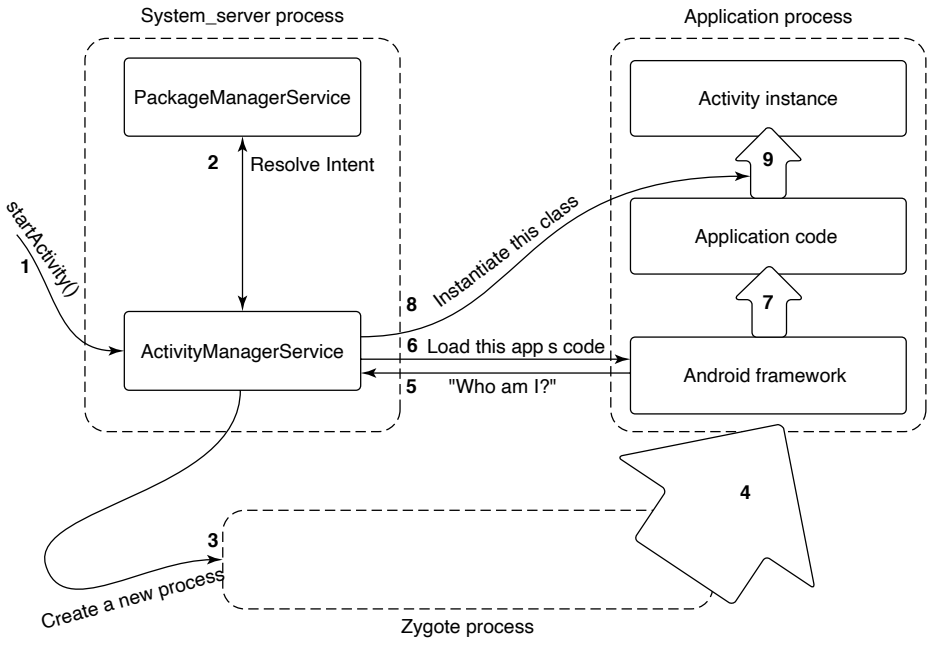

Figure 10-62. Steps in launching a new application process.

Note that when we started this activity, the application's process may already have been running. In that case, the activity manager will simply skip to the end, sending a new command to the process telling it to instantiate and run the appropriate component. This can result in an additional activity instance running in the application, if appropriate, as we saw previously in Fig. 10-56.

Process Lifecycle

The activity manager is also responsible for determining when processes are no longer needed. It keeps track of all activities, receivers, services, and content providers running in a process; from this it can determine how important (or not) the process is.

Recall that Android's out-of-memory killer in the kernel uses a process's importance as given to *lmkd* as a strict ordering to determine which processes it

should kill first. The activity manager is responsible for setting each process's importance appropriately based on the state of that process, by classifying them into major categories of use. Figure 10-63 shows the main categories, with the most important category first. The last column shows a typical importance value that is assigned to processes of this type.

Category	Description	Importance
SYSTEM	The system and daemon processes	–900
PERSISTENT	Always-running application processes	–800
FOREGROUND	Currently interacting with user	0
VISIBLE	Visible to user	100–199
PERCEPTIBLE	Something the user is aware of	200
SERVICE	Running background services	500
HOME	The home/launcher process (when not in foreground)	600
CACHED	Processes not in use	950–999

Figure 10-63. Process importance categories.

Now, when RAM is getting low, the system has configured the processes so that the out-of-memory killer will first kill *cached* processes to try to reclaim enough needed RAM, followed by *home*, *service*, and on up. Within a specific importance level, it will kill processes with a larger RAM footprint before smaller ones.

We've now seen how Android decides when to start processes and how it categorizes those processes in importance. Now we need to decide when to have processes exit, right? Or do we really *need* to do anything more here? The answer is, we do not. On Android, *application processes never cleanly exit*. The system just leaves unneeded processes around, relying on the kernel to reap them as needed.

Cached processes in many ways take the place of the swap space that Android lacks. As RAM is needed elsewhere, cached processes can be killed and their RAM quickly reclaimed. If an application later needs to run again, a new process can be created, restoring any previous state needed to return it to how the user last left it. Behind the scenes, the operating system is launching, killing, and relaunching processes as needed so the important foreground operations remain running and cached processes are kept around as long as their RAM would not be better used elsewhere.

Process Dependencies

We now have a good overview of how individual Android processes are managed. There is a further complication to this, however: dependencies between processes. Processes caninteract with other processes and that has to be managed.

As an example, consider our previous camera application holding the pictures that have been taken. These pictures are not part of the operating system; they are implemented by a content provider in the camera application. Other applications may want to access that picture data, becoming a client of the camera application.

Dependencies between processes can happen with both content providers (through simple access to the provider) and services (by binding to a service). In either case, the operating system must keep track of these dependencies and manage the processes appropriately.

Process dependencies impact two key things: when processes will be created (and the components created inside of them), and what the importance of the process will be. Recall that the importance of a process is that of the most important component in it. Its importance is also that of the most important process that is dependent on it.

For example, in the case of the camera application, its process and thus its content provider is not normally running. It will be created when some other process needs to access that content provider. While the camera's content provider is being accessed, the camera process will be considered at least as important as the process that is using it.

To compute the final importance of every process, the system needs to maintain a dependency graph between those processes. Each process has a list of all services and content providers currently running in it. Each service and content provider itself has a list of each process using it. (These lists are maintained in records inside the activity manager, so it is not possible for applications to lie about them.) Walking the dependency graph for a process involves walking through all of its content providers and services and the processes using them.

Figure 10-64 illustrates a typical state processes can be in, taking into account dependencies between them. Part of this example contains two dependencies, where a content provider in a camera app is being used by a separate email app to add a picture attachment. (An illustration of this situation appears later in Fig. 10-70 and is discussed in more detail there.)

In this figure, after the regular system processes, is first that current foreground email application. The email application is making use of the camera content provider, raising the camera process up to the same importance as the email app. Next in the figure is a similar situation, a music application is playing music in the background with a service, and while doing so has a dependency on the media process for accessing the user's music media, which similarly raises the media process up to the same importance as the music app.

Consider what happens if the state of Fig. 10-64 changes so that the email application is done loading the attachment, and no longer uses the camera content provider. Figure 10-65 illustrates how the process state will change. Note that the camera application is no longer needed, so it has dropped out of the foreground importance, and down to the cached level. Making the camera cached has also pushed the old maps application one step down in the cached LRU list.

Process	State	Importance
system	Core part of operating system	SYSTEM
phone	Always running for telephony stack	PERSISTENT
email	Current foreground application	FOREGROUND
camera	In use by email to load attachment	FOREGROUND
music	Running background service playing music	PERCEPTIBLE
media	In use by music app for accessing user's music	PERCEPTIBLE
download	Downloading a file for the user	SERVICE
launcher	App launcher not current in use	HOME
maps	Previously used mapping application	CACHED

Figure 10-64. Typical state of process importance.

Process	State	Importance
system	Core part of operating system	SYSTEM
phone	Always running for telephony stack	PERSISTENT
email	Current foreground application	FOREGROUND
music	Running background service playing music	PERCEPTIBLE
media	In-use by music app for accessing user's music	PERCEPTIBLE
download	Downloading a file for the user	SERVICE
launcher	App launcher not current in use	HOME
camera	Previously used by email	CACHED
maps	Previously used mapping application	CACHED+1

Figure 10-65. Process state after email stops using camera.

These two examples give a final illustration of the importance of cached processes. If the email application again needs to use the camera provider, the provider's process will typically already be left as a cached process. Using it again is then just a matter of setting the process back to the foreground and reconnecting with the content provider that is already sitting there with its database initialized.

10.8.11 Security and Privacy

When Android was being designed, the security protections users have from their applications was an area of rapidly evolving expectations that needed to be addressed. Since then, privacy has become an increasingly important area driving significant evolution to how Android manages applications. We will now look at these two topics, focusing first on the various aspects of security before looking at the newer world of privacy.

Application Sandboxes

Traditionally in operating systems, applications are seen as code executing as the user, on the user's behalf. This behavior has been inherited from the command line, where you run the `ls` command and expect that to run as your identity (UID), with the same access rights as you have on the system. In the same way, when you use a graphical user interface to launch a game you want to play, that game will effectively run as your identity, with access to your files and many other things it may not actually need.

This is not, however, how we mostly use computers today. We run applications we acquired from some less trusted third-party source, and those apps can have sweeping functionality, doing a wide variety of things that we have little control over. There is a disconnect between the application model supported by the operating system and the one actually in use. This may be mitigated by strategies such as distinguishing between normal and "admin" user privileges and issuing a warning the first time an application runs, but those do not really address the underlying disconnect.

In other words, traditional operating systems are very good at protecting users from other users, but not at protecting users from themselves and their applications. All programs run with the power of the user and, if any of them misbehaves, it can do all the same damage as the user (and sometimes more). Think about it: how much damage could you do in, say, a UNIX environment? You could leak all information accessible to the user. You could perform `rm –rf *` to give yourself a nice, empty home directory. And if the program is not just buggy, but also malicious, it could encrypt all your files for ransom. Running everything with "the power of you" is dangerous!

On mobile devices at the time Android was being developed, this problem of protecting users from their applications was typically addressed by the introduction of a gatekeeper to the device: one or more trusted entities (such as the telecommunications carrier or manufacturer of the device) who are responsible for determining whether an application is safe before allowing it to be installed. Such an approach was counter to a key goal of Android, to create an open platform where everyone could compete equally and there was no single entity controlling what the user could do on their device, so another solution was needed.

Android addresses the problem with a core premise: that an application is actually the developer of that application running as a guest on the user's device. Thus, an application is not trusted with anything sensitive that is not explicitly approved by the user.

In Android's implementation, this philosophy is rather directly expressed through user IDs. When an Android application is installed, a new unique Linux user ID (or UID) is created for it, and all of its code runs as that "user." Linux user IDs thus create a sandbox for each application, with their own isolated area of the file system, just as they create sandboxes for users on a desktop system. In other

words, Android uses an existing core feature in Linux, but in a novel way. The result is better isolation.

Application security in Android revolves around UIDs. In Linux, each process runs as a specific UID, and Android uses the UID to identify and protect security barriers. The only way to interact across processes is through some IPC mechanism, which generally carries with it enough information to identify the UID of the caller. Binder IPC explicitly includes this information in every transaction delivered across processes so a recipient of the IPC can easily ask for the UID of the caller.

Android predefines a number of standard UIDs for the lower-level parts of the system, but most applications are dynamically assigned a UID, at first boot or install time, from a range of "application UIDs." Figure 10-66 illustrates some common mappings of UID values to their meanings. UIDs below 10000 are fixed assignments within the system for dedicated hardware or other specific parts of the implementation; some typical values in this range are shown here. In the range 10000–19999 are UIDs dynamically assigned to applications by the package manager when it installs them; this means at most 10,000 applications can be installed on the system. Also note the range starting at 100000, which is used to implement a traditional multiuser model for Android: an application that is granted UID 10002 as its identity would be identified as 110002 when running as a second user.

UID	Purpose
0	Root
1000	Core system (system_server process)
1001	Telephony services
1013	Low-level media processes
2000	Command line shell access
10000–19999	Dynamically assigned application UIDs
100000	Start of secondary users

Figure 10-66. Common UID assignments in Android.

When an application is first assigned a UID, a new storage directory is created for it, with the files there owned by its UID. The application gets full access to its private files there, but cannot access the files of other applications, nor can the other applications touch its own files. This makes content providers, as discussed in the earlier section on applications, especially important, as they are one of the few mechanisms that can transfer data between applications.

Even the system itself, running as UID 1000, cannot touch the files of applications. This is why the *installd* daemon exists: it runs with special privileges to be able to access and create files and directories for other applications. There is a very restricted API *installd* provided to the package manager for it to create and manage the data directories of applications as needed.

Permissions

In their base state, Android's application sandboxes must disallow any cross-application interactions that can violate security between them. This may be for robustness (preventing one app from crashing another app), but most often it is about information access.

Consider our camera application. When the user takes a picture, the camera application stores that picture in its private data space. No other applications can access that data, which is what we want since the pictures there may be sensitive data to the user.

After the user has taken a picture, she may want to email it to a friend. Email is a separate application, in its own sandbox, with no access to the pictures in the camera application. How can the email application get access to the pictures in the camera application's sandbox?

The best-known form of access control in Android is application permissions. Permissions are specific well-defined abilities that can be granted to an application at install time. The application lists the permissions it needs in its manifest, and depending on the type of permission they will either be granted at install time (if allowed) or can ask the user to grant them the permission while running.

Figure 10-67 shows how our email application could make use of permissions to access pictures in the camera application. In this case, the camera application has associated the READ_PICTURES permission with its pictures, saying that any application holding that permission can access its picture data. The email application declares in its manifest that it requires this permission. The email application can now access a URI owned by the camera, such as content://pics/1; upon receiving the request for this URI, the camera app's content provider asks the package manager whether the caller holds the necessary permission. If it does, the call succeeds and appropriate data are returned to the application.

Permissions are not tied to content providers; any IPC into the system may be protected by a permission by asking the package manager if the caller holds the required permission. Recall that application sandboxing is based on processes and UIDs, so a security barrier always happens at a process boundary, and permissions themselves are associated with UIDs. Given this, a permission check can be performed by retrieving the UID associated with the incoming IPC and asking the package manager whether that UID has been granted the corresponding permission. For example, permissions for accessing the user's location are enforced by the system's location manager service when applications call in to it.

Figure 10-68 shows what happens when an application does not hold a permission needed for an operation it is performing. Here the browser application is trying to directly access the user's pictures, but the only permission it holds is one for network operations over the Internet. In this case the PicturesProvider is told by the package manager that the calling process does not hold the needed READ_PICTURES permission, and as a result throws a SecurityException back to it.

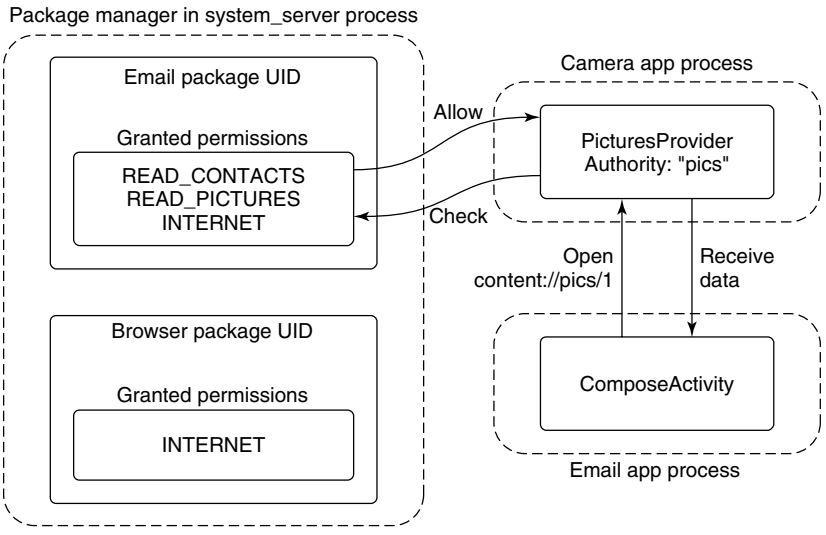

Figure 10-67. Requesting and using a permission.

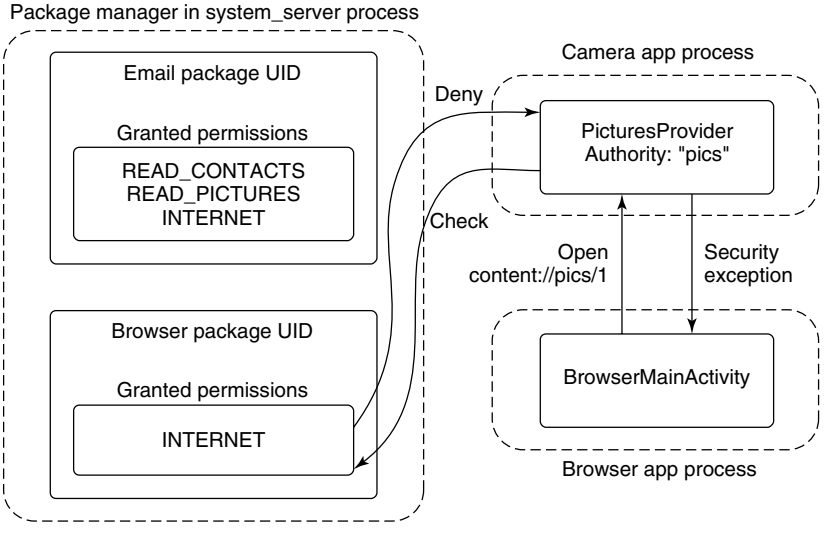

Figure 10-68. Accessing data without a permission.

Permissions provide broad, unrestricted access to classes of operations and data. They work well when an application's functionality is centered around those operations, such as our email application requiring the INTERNET permission to

send and receive email. However, does it make sense for the email application to hold a READ_PICTURES permission? There is nothing about an email application that is directly related to reading the user's pictures, and no reason for an email application to have access to all of those pictures.

There is another issue with this use of permissions, which we can see by returning to Fig. 10-55. Recall how we can launch the email application's ComposeActivity to share a picture from the camera application. The email application receives a URI of the data to share, but does not know where it came from—in the figure here it comes from the camera, but any other application could use this to let the user email its data, from audio files to word-processing documents. The email application only needs to read that URI as a byte stream to add it as an attachment. However, with permissions it would also have to specify up-front the permissions for all of the data of all of the applications it may be asked to send an email from.

We have two problems to solve. First, we do not want to give applications access to wide swaths of data that they do not really need. Second, they need to be given access to any data sources, even ones they do not have a priori knowledge about.

There is an important observation to make: the act of emailing a picture is actually a user interaction where the user has expressed a clear intent to use a specific picture with a specific application. As long as the operating system is involved in the interaction, it can use this to identify a specific hole to open in the sandboxes between the two applications, allowing that data through.

Android supports this kind of implicit secure data access through intents and content providers. Figure 10-69 illustrates how this situation works for our picture emailing example. The camera application at the bottom-left has created an intent asking to share one of its images, content://pics/1. In addition to starting the email compose application as we had seen before, this also adds an entry to a list of "granted URIs," noting that the new ComposeActivity now has access to this URI. Now when ComposeActivity looks to open and read the data from the URI it has been given, the camera application's PicturesProvider that owns the data behind the URI can ask the activity manager if the calling email application has access to the data, which it does, so the picture is returned.

This fine-grained URI access control can also operate the other direction. An example here is another intent action, android.intent.action.GET_CONTENT, which an application can use to ask the user to pick some data and return it back. This would be used in our email application, for example, to operate the other way around: the user while in the email application can ask to add an attachment, which will launch an activity in the camera application for them to select one.

Figure 10-70 shows this new flow. It is almost identical to Fig. 10-69, the only difference being in the way the activities of the two applications are composed, with the email application starting the appropriate picture-selection activity in the camera app. Once an image is selected, its URI is returned back to the email application, and at this point our URI grant is recorded by the activity manager.

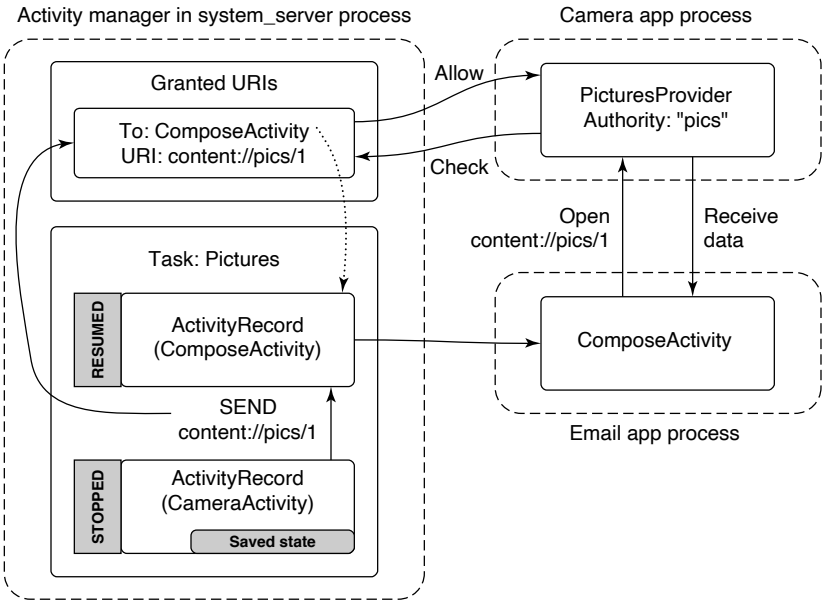

Figure 10-69. Sharing a picture using a content provider.

This approach is extremely powerful, since it allows the system to maintain tight control over per-application data, granting specific access to data where needed, without the user needing to be aware that this is happening. Many other user interactions can also benefit from it. An obvious one is drag and drop to create a similar URI grant, but Android also takes advantage of other information such as current window focus to determine the kinds of interactions applications can have.

A final common security method Android uses is explicit user interfaces for allowing/removing specific types of access. In this approach, there is some way an application indicates it can optionally provide some functionally, and a system-supplied trusted user interface that provides control over this access.

A typical example of this approach is Android's input-method architecture. An input method is a specific service supplied by a third-party application that allows the user to provide input to applications, typically in the form of an on-screen keyboard. This is a highly sensitive interaction in the system, since a lot of personal data will go through the input-method application, including passwords the user types.

An application indicates it can be an input method by declaring a service in its manifest with an intent filter matching the action for the system's input-method protocol. This does not, however, automatically allow it to become an input method, and unless something else happens the application's sandbox has no ability to operate like one.

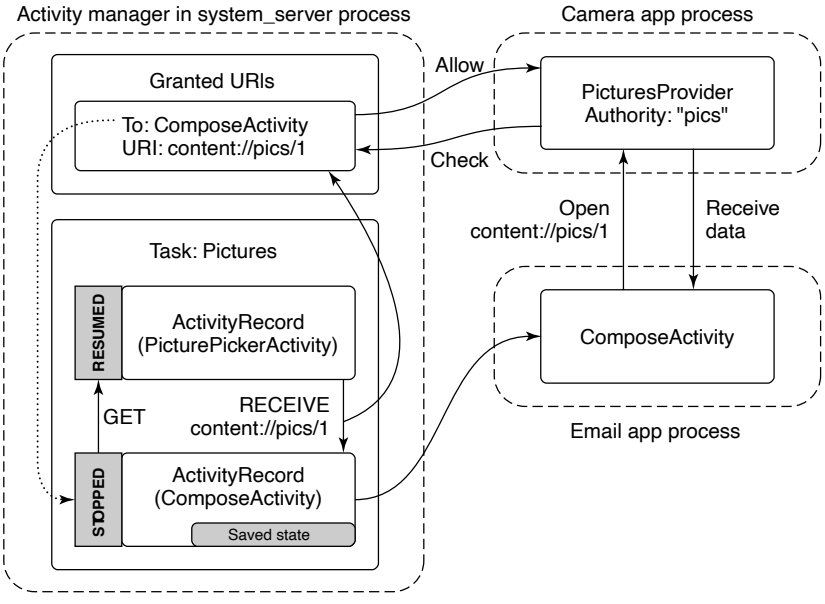

Figure 10-70. Adding a picture attachment using a content provider.

Android's system settings include a user interface for selecting input methods. This interface shows all available input methods of the currently installed applications and whether or not they are enabled. If users want to use a new input method after they have installed the application, they must go to this system settings interface and enable it. When doing that, the system can also inform the user of the kinds of things this will allow the application to do.

Even once an application is enabled as an input method, Android uses fine-grained access-control techniques to limit its impact. For example, only the application that is being used as the current input method can actually have any special interaction; if the user has enabled multiple input methods (such as a soft keyboard and voice input), only the one that is currently in active use will have those features available in its sandbox. Even the current input method is restricted in what it can do, through additional policies such as only allowing it to interact with the window that currently has input focus.

SELinux and Defense in Depth

A robust security architecture is important: one where access to data is minimized, the architecture is easy to understand so that it is less likely for bugs to be introduced during development, and changes that violate the intended security guarantees are easy to identify. Even in the best design, however, bugs will always

happen, resulting in significant security issues that are shipped and need to be fixed. It is thus also important to adopt a "defense in depth" strategy to minimize the impact of a single security bug.

Sandboxing forms the foundation of Android's security architecture and defense-in-depth approach. For example, Android provides a special kind of UID sandbox called an "isolated service." This is a service that runs in its own dedicated process, with a transient UID that is not associated with any capabilities: no access to any permissions, or most system services, or app filesystem, etc. This facility is used to render things like Web pages and PDF files, content that is extremely complicated to handle and thus often has bugs that allow such content, retrieved from an untrusted source, to deliver an exploit through bugs in the content handling code.

Since the capabilities of an isolated process are minimized, exploits in that content often need to find a security hole in both the isolated sandbox that allow it to get out to the app sandbox, and then a hole in the app sandbox to exploit the system itself.

This restricted sandbox approach is used throughout Android. Of particular note is the media system, which initially suffered a significant number of exploits (given the name "stagefright" from the name of the core media library). Like Web pages and PDFs, media codecs deal with complicated formats of data that comes from untrusted sources, making them ripe for exploit. The solution here was to likewise isolate these codecs and other parts of the media system into highly restricted sandboxes that only gave them the capabilities needed for their operation and nothing more.

Sandboxes do have limitations: their functionality, though limited, is still fairly significant. Vulnerabilities in the things they interact with (especially the kernel) can allow them to bypass most of the system's security. In Android 5.0, SELinux was introduced as an additional security layer in the platform that works in conjunction with its existing UID-based sandboxes as well as providing more fine-grained sandboxing for system components.

The security mechanisms we have talked about so far use a model called discretionary access control (DAC), meaning the entity creating a resource (such as a file) has the discretion to determine who has access to it. SELinux, in contrast, provides mandatory access control (MAC), meaning all access to resources is defined statically and separately from the code. In SELinux, an entity starts without access to anything, and rules are written to explicitly specify what it is allowed to do.

SELinux by itself cannot be used to implement Android's security model, because it is not flexible enough: it would not allow one application to get access to a piece of data from another application only when the user says that is allowed. Rather, SELinux provides a parallel security mechanism with different capabilities and benefits. While some security restrictions are enforced via only UID or SELinux, where possible Android will utilize both mechanisms to provide defense-in-depth for security restrictions.

As an example of what SELinux provides, consider a simple bug where some system code writes a file and accidentally makes it world readable, such as a file keeping track of the permissions granted to apps. In the UID-based security model, this mistake allows any app sandbox to modify this file, such as to change it to say it has a permission the user did not actually give it.

With SELinux enabled, however, this exploit is defeated: Android's SELinux rules say that no app sandbox can read or write a system file, so the exploit will still be stopped. Each UID sandbox also has an associated SELinux context defining the rules for what it is allowed to do, written to be as minimal as possible. For example, the rules for an isolated service's sandbox say that it has no read/write access at all to data files.

More information on how Android uses SELinux can be found online at *https://source.android.com/security/selinux*.

Privacy and Permissions

Privacy is a newer but increasingly important issue that operating systems must address. Where security can be described as addressing the goal that "nothing placed on the device can harm it or the user" (such as harm its operation, force the user to pay money to access it, force ads on users, allow other apps to be installed they do not want, etc.), the goal of privacy is to help users be confident that "the information about them is being protected and only used for what they want."

Security is most notable to the user in its absence: if the device's security is good, it always behaves as intended and the user never has a bad experience from malware. Privacy, in contrast, involves a more direct interaction between the operating system and the users, because it requires that they have confidence that the platform is looking out for their data, allows them to make the decisions they want about how their data is protected, and gives some visibility into what happens to their data.

To help illustrate the difference between security and privacy, consider Fig. 10-71, which the only thing most users want to know about the security of their operating system (if even that). Keep this in mind as we look at the thinking that goes behind designing the privacy of the system.

Privacy cannot happen without security: without a secure foundation for controlling what apps can do, an operating system cannot give assurance about what happens to the users' data—a malicious app could access their data through insecure paths without the user knowing. And though security on Android provides the walls that allow statements about privacy to have meaning, security is not by itself sufficient to address privacy concerns.

When Android was first designed, security was the primary focus for its users and developers: operating systems were still evolving to address security in the modern world of wide-spread use of devices that allow people to install and use apps without concern for them causing damage. Mobile devices further exacerbated security issues due the increased personal nature of them, such as always

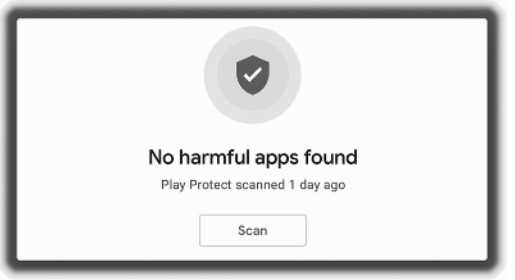

Figure 10-71. The only thing most users care about security.

being with someone and thus always having potential access to sensitive information such as their location. This makes the evolution of Android around privacy an interesting case-study in how these issues have been evolving in the industry.

Android's initial approach to privacy was security-focused: every application needed to declare in its manifest the sensitive data and capabilities it needed access to, and the platform strictly enforced this. The user experience revolved around showing the users what the app would have access to before it was installed, allowing them to decide if they were okay with it having that information before going forward to install (and with confidence it would not get any other information once installed). An example of this user experience is shown in Fig. 10-72.

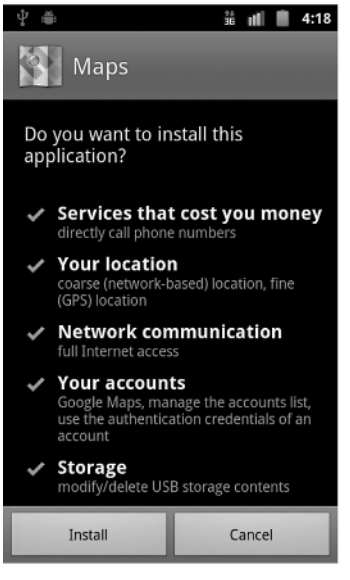

Figure 10-72. Confirming permissions at install time (circa 2010)

There were a wide variety of permissions, organized into categories to help users understand the major classes of operations the app may do. A summary of these permissions and their categories is shown in Fig. 10-73. The permissions listed here are all *dangerous* permissions, meaning they were considered important enough to always show to users to let them decide whether to proceed with an install.

Permission	Group
SEND_SMS	COST_MONEY
CALL_PHONE	COST_MONEY
RECEIVE_SMS	MESSAGES
READ_SMS	MESSAGES
WRITE_SMS	MESSAGES
READ_CONTACTS	PERSONAL_INFO
WRITE_CONTACTS	PERSONAL_INFO
READ_CALENDAR	PERSONAL_INFO
WRITE_CALENDAR	PERSONAL_INFO
BODY_SENSORS	PERSONAL_INFO
ACCESS_FINE_LOCATION	LOCATION
ACCESS_COARSE_LOCATION	LOCATION
INTERNET	NETWORK
BLUETOOTH	NETWORK
MANAGE_ACCOUNTS	ACCOUNTS
MODIFY_AUDIO_SETTINGS	HARDWARE_CONTROLS
RECORD_AUDIO	HARDWARE_CONTROLS
CAMERA	HARDWARE_CONTROLS
PROCESS_OUTGOING_CALLS	PHONE_CALLS
MODIFY_PHONE_STATE	PHONE_CALLS
READ_PHONE_STATE	PHONE_CALLS
WRITE_SETTINGS	SYSTEM_TOOLS
SYSTEM_ALERT_WINDOW	SYSTEM_TOOLS
WAKE_LOCK	SYSTEM_TOOLS
READ_EXTERNAL_STORAGE	STORAGE
WRITE_EXTERNAL_STORAGE	STORAGE

Figure 10-73. Select list of install-time dangerous permissions.

There were an additional set of *normal* permissions, which the application still needed to request in its manifest to be able to use, but would only be shown to the users if they explicitly asked to see more details before installing. A representative list of these permissions is shown in Fig. 10-74. Note for example that access to

the camera and microphone is protected by dangerous permissions, above, since these give access to sensitive personal data; access to the vibration hardware and flashlight are normal since the worst the app can do with this is annoy the user.

Permission	Group
SET_ALARM	SET_ALARM
ACCESS_NETWORK_STATE	NETWORK
ACCESS_WIFI_STATE	NETWORK
GET_ACCOUNTS	ACCOUNTS
VIBRATE	HARDWARE_CONTROLS
FLASHLIGHT	HARDWARE_CONTROLS
EXPAND_STATUS_BAR	SYSTEM_TOOLS
KILL_BACKGROUND_PROCESSES	SYSTEM_TOOLS
SET_WALLPAPER	SYSTEM_TOOLS

Figure 10-74. Select list of install-time normal permissions.

Android 6.0 switched the user's permission experience from the previous install-time model to a runtime model. This means that instead of granting the application a permission's capabilities at the point of install, for many permissions the app now must explicitly ask the user at runtime through a system prompt as illustrated in Fig. 10-75.

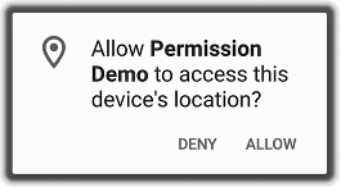

Figure 10-75. Android 6.0 runtime permission prompt.

Moving to runtime prompts could not simply take the existing permissions as is and present them to the user one at a time, while the app is running, as it needs them: that would be overwhelming to the user. It thus required extensive rework of the permission organization so they are appropriate for runtime permissions, resulting in the new model shown in Fig. 10-76.

The permissions here (now on the right side of the table) are still classified as dangerous permissions, but not directly shown to users; rather, the group they are in (on the right side) is the runtime prompt that will be shown to the user, allowing the app to get access to all permissions it has requested in that group. The granularity of the underlying permissions is thus retained, but the amount of information and choice the user must deal with is greatly decreased.

Runtime prompt	Permissions
CONTACTS	READ_CONTACTS, WRITE_CONTACTS, GET_ACCOUNTS
CALENDAR	READ_CALENDAR, WRITE_CALENDAR
SMS	SEND_SMS, RECEIVE_SMS, READ_SMS
STORAGE	READ_EXTERNAL_STORAGE, WRITE_EXTERNAL_STORAGE
LOCATION	ACCESS_FINE_LOCATION, ACCESS_COARSE_LOCATION
PHONE	READ_PHONE_STATE, CALL_PHONE, PROCESS_OUTGOING_CALLS
MICROPHONE	RECORD_AUDIO
CAMERA	CAMERA
SENSORS	BODY_SENSORS

Figure 10-76. Select list of runtime permissions.

There are still normal permissions, but they are no longer shown to the user at all. Instead, the platform still restricts access to them, so that information in the manifest can be used to audit applications with guarantees about what they can and cannot do on the device. The remaining permissions from before that are now auditable normal permissions are shown in Fig. 10-77.

Permission
SET_ALARM
ACCESS_NETWORK_STATE
ACCESS_WIFI_STATE
VIBRATE
FLASHLIGHT
EXPAND_STATUS_BAR
KILL_BACKGROUND_PROCESSES
SET_WALLPAPER
INTERNET
BLUETOOTH
MODIFY_AUDIO_SETTINGS
WAKE_LOCK

Figure 10-77. Select list of auditable normal permissions.

This organizational change effectively moved the permission design from security-centric to privacy-centric. The new permission groups represent separate types of data the user may be interested in protecting, and everything else has been hidden from them.

For something to justify being shown as a runtime permission, it must clearly pass a test: "Is this something the user easily understands (which generally means it represents some clear data about them), and can be confident in making a decision about releasing access to that data?" Users answering yes to a runtime permission prompt is them making a statement that they are going to trust that app (and its developer) with all of that type of personal data on their device.

The INTERNET permission is a good case study in this design process: it was modified from a dangerous permission shown to the user at install, to a normal permission that does not require a runtime prompt and is never shown to the user. The reasoning behind this is given below:

1. **How many applications would ask for this as a runtime permission?** Most of them, so the user will be confronted with it frequently and needs to be especially confident about making a good decision. (Frequent prompts for decisions the user is not confident in can easily lead to all of the prompts being mostly ignored by them.)
2. **Is this protecting some data the user can clearly understand?** No. That makes it harder for the user to understand what is being asked.
3. **Is this giving the application an ability the user cares about?** Yes. In a way, apps being able to access the network seems like something that is of interest to the user's privacy.
4. **Why would a user decide whether or not to give an app the permission?** A common thought process here is: "I do not want the app to access the network so it cannot send my data off the device."
5. Deciding to allow access to the network actually has a close connection to decisions around giving it access to personal data! That is, a user saying "no" to the network permission will often lead to them feeling better about saying "yes" to requests to get access to their data.
6. Wanting to control network access is thus actually a proxy for wanting a guarantee about the app not being able to export any data off the device. However, *that is not what the network permission does*. Even if an app does not have network access, there are many ways it can export data, even accidentally: for example if it opens the browser on a Website associated with it, the URL it hands to the browser can contain any data it wants, which is then sent to the app's server.

It is best that network access not be a runtime permission, for multiple reasons. It would be requested by most apps, causing the user to be constantly confronted with it. They are being asked to make a decision that is not clear how it impacts them. The main reason that many users would infer why they should say

"no"—that it prevents the app from exporting data—can lead them to make bad decisions for other permissions the app requests. The last point compromises the fundamental permission model: that saying "yes" to a permission prompt is expressioning trust in the app with that data.

There are, finally, a few permissions that completely disappeared in the runtime mode, such as WRITE_SETTINGS and SYSTEM_ALERT_WINDOW. Typically these were deemed too dangerous to just hide or even have as a simple runtime prompt (or too hard to understand for the user to make a good decision in a simple runtime prompt). Typically these were transformed into an explicit user interface that the user must go in to manually enable access of the app to that permission, as covered previously when discussing permissions and explicit user interfaces for controlling them.

This then provides a basic framework for deciding how a particular feature in the platform will be secured, in a privacy-oriented way:

1. If it can be done as part of a larger user flow, where the users do not realize they are making a security/privacy decision, that is ideal. Examples of such flows are the URI permission grants driven by share and android.intent.action.GET_CONTENT experiences described previously.
2. If it is something that does not significantly impact the user's privacy or put the device at risk, a normal auditable permission is a good choice.
3. If it is associated with clear personal data, the user is likely to have a strong opinion about who can access it, so a runtime permission is probably a good choice.
4. Otherwise, it may need to be a separate explicit user interface for giving only certain apps that specific privilege. The more dangerous this is to the user, however, the more carefully it must be done. For example, the WRITE_SMS permission was changed to a separate interface where it is only given to one app that a user can designate as the preferred text messaging app. This helps everyone make a safer decision by instead thinking about which app should get this feature.

Evolving Runtime Permissions

The move to runtime permissions was only the start of Android's privacy journey, which will continue to be a core design consideration for operating systems just like security. To illustrate these changes, we will look specifically at the location permission and how it evolved over later Android releases.

Recall that in Android 6.0, the user experience for location access shown in the previous Fig. 10-75 was a simple "yes" or "no" question, hiding even the difference between coarse and fine grained location access, to create a simple experience. This provided significant new control for users, but as the ability to access the user's location increasingly became a point of concern (both due to increased user awareness and increased problematic use by apps), demands for more control drove a series of changes from the initial simple runtime permission.

The first change to location access was invisible to users: in Android 8.0 the concept of background vs. foreground location access was introduced. When an application is considered to be in the background, it is not able to get location updates at a high rate.

The motivation for this was partly to improve the battery life of Android devices, since applications constantly monitoring location in the background could consume significant power, but it also reduced the amount of information about the user that these apps could collect. (Applications that really need to closely monitor location while in the background can do this through the use of foreground services, which are discussed later in Background Execution.)

Android 10 took a more privacy-centric approach to this problem, making the difference between background and foreground location access an explicit part of the user's experience. This was presented to the user in the form of a new runtime prompt, shown in Fig. 10-78, where the user could select the kind of access the app should have.

Figure 10-78. Android 10's background vs. foreground location prompt.

Driven by growing demands for more privacy, this new permission prompt is the first time the platform used the concept of background vs. foreground execution of apps in its core user experience. Note the careful wording here: foreground is described as "only while the app is in use" and background is "all the time," reflecting the actual underlying complexity of these concepts. For example, if you are currently using a mapping application to do navigation but are not actively in the app on the screen, is it considered foreground or background? From Android's

perspective it is foreground for location access, but "while the app is in use" better explains this to the user.

Android 11 went a step further and introduced a new concept of "only this once," shown in Fig. 10-79, now giving the user an option to restrict location access to only their current session in the app. When selected, once the app is exited, the location permission will be silently revoked and cause the app to no longer have location access. The next time the app is used, there will be another prompt for location access and the user can decide in this new situation what to allow.

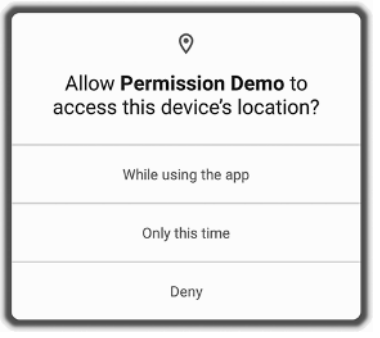

Figure 10-79. Android 11's "only this once" prompt.

A transient permission grant is useful for permissions like location, where any time apps have access to it they have available a continuous stream of new personal data about the user, in this case where the user is located. (The same capability was at this time applied to two other permissions with similar semantics, access to the camera and microphone.) This addresses the situation where users feels like the app is asking for access to such data in a situation that makes sense now, but they do not think the app normally needs that access.

Note also another change to the location experience, where the option to give background access to location is completely gone. This happened because having more than three options results in an overly complicated experience for users trying to decide what they want, and the vast majority of applications do not need full background access since most such use cases are served better by foreground services.

For the rare cases where an app really could make use of full background location access, and the user can be convinced to allow this, the option still remains in the overall system settings for the app's permissions, shown in Fig. 10-80. Here the user can see all of the possible options, including the option currently selected for the app (if any), and change the selection as desired.

Most recently, Android 12 further extends the options available to the user about location access by giving them the option to select between coarse vs. fine

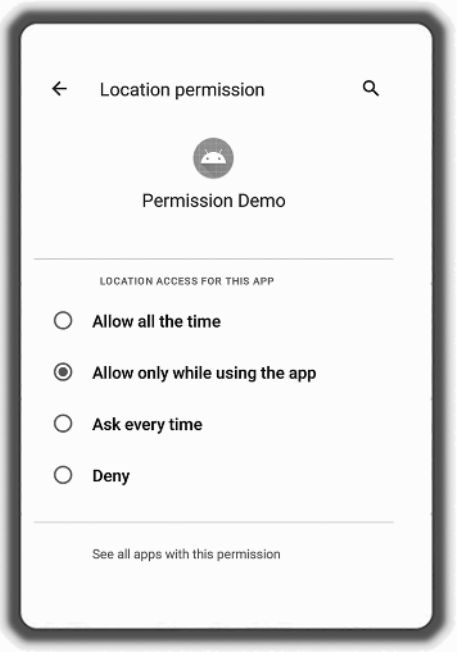

Figure 10-80. Android 11's location permission settings.

access as shown in Fig. 10-81. Note that these are essentially the same types of location access applications and the user could differentiate between back at the start in Android 1.0! They were hidden from the user, but were still options for the app, in Android 5.0. Android 12 again shows them explicitly to the user while also allowing them to override the app's preference (if it is requesting fine access).

Android 12 also introduced a new "privacy dashboard," allowing users to see when apps are accessing their location and other personal data after they have granted that access. Fig. 10-82 shows an example of what a user may see about location access across their device. This provides a rich tool for users to monitor what their apps are doing, to reassure themselves they are comfortable with it and potentially change their decision about an app's access based on what they see.

The changes we have discussed (from the transition to runtime permissions, through evolution of location access, to privacy dashboard) all serve to illustrate how privacy has become a unique aspect of operating system design. Most operating system features are better the less the user is aware of them. This is true not only for security, as we previously described, but generally the better solution for a problem is one where the operating system can do something so that the user does not need to think about it. We saw another example of this earlier, with Android removing the need for users to think about explicitly starting and stopping their apps.

Figure 10-81. Android 12's coarse vs. fine prompt.

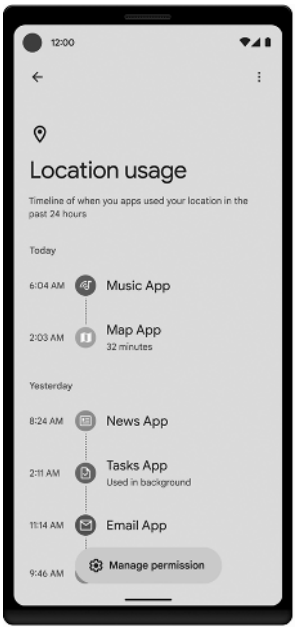

Figure 10-82. Android 12's privacy dashboard showing "location" details.

Privacy, in contrast, is a collaboration with the user, gaining their trust by clearly informing them of what is happening with their data and providing controls for them to express their preferences. It is hard for an operating system to do this automatically, not only because having this information and control is central to gaining trust, but also because there is no right set of answers for all users: if you survey users about their preferences for how their data is handled, some will care much less than others (caring more about features they get by providing their data), and some will have significantly stronger preferences for certain types of data compared to others with strong preferences for different data.

10.8.12 Background Execution and Social Engineering

One of Android's initial design goals was to create an open mobile operating system, allowing regular app developers the flexibility to not only implement much of the same functionality as provided by its built-in applications, but also to create new kinds of applications not originally envisioned by the platform.

This design goal was expressed in the previously covered application model of activities, receivers, services, and content providers: a set of flexible basic building blocks applications use to express their needs to the operating system. Of special note is the service, a general mechanism for an app to express the need to do some work in the background even if the user is not currently running their app.

A service can represent a wide range of functionality, from various kinds of updating and syncing data in the background, to more explicitly user-controlled execution. For example, Android shipped with a music player that allowed the user to continue listening to music even while not in the application itself. Since this could be built with the basic service construct, from the first version of Android any regular application could implement that same functionality, and even use it for entirely new kinds of experiences such as driving navigation or exercise tracking.

Android's flexibility in background execution was valuable, but also became an increasing challenge to manage, which this section will look at in more detail. But before doing that, let's consider a simple case of foreground services.

A foreground service is a capability for a running service component to tell Android that it is especially important to the user. This gives the system an important distinction between more important and less important services, for things like memory management. Recall Fig. 10-63 showing different process importance categories. Whether a service is foreground or not determines whether its process is classified as *perceptible* or *service*. By being more important than regular services (but less important than the visible application), Android can correctly decide to get rid of processes for background services without breaking experiences like the user listening to music in the background.

In Android 1.0, a services was made foreground with a simple API that directly requested it, and the system trusted that apps used this for the intended purpose:

something the *user is aware of* like background music playback. However, soon after 1.0 shipped, it was observed that applications would often use the API incorrectly, setting something to be foreground that was not really so important to the user. This behavior started to cause bad experiences for users, as the services they did care about would get killed due to services they did not.

The foreground service issue was addressed in Android 2.0 by requiring that, in order to make a service foreground, it also needs to have an ongoing notification associated with it. This tied the purpose of a foreground service (doing something the user is directly aware of) to something an app would only want to do in such a situation (inform the user about what it is doing in a very visible way). Playing music in the background, navigating with maps, tracking exercise—all of these things naturally involve displaying a notification so the user can easily see what is happening and control it, even when not in the app that is doing the operation.

Though the notification solution worked well in incentivizing developers to use foreground services for their intended purpose, over time a more general issue of apps running in the background became an growing problem for Android that needed to be further addressed. To understand why, let's consider the way an operating system like Android deals with a limited resource such as battery power.

The battery of a mobile device is an important, limited resource. For each charge of the battery, you can get a fixed amount of work done. People expect their battery to last through a normal day without needing to be charged, so there is a fixed amount of work that a device can do each day. Ideally the battery only drains while its screen is on and in use, so there is a fairly clear amount of actual work you can use the device for each day. However, while the screen is off numerous things can also consume power, such as:

1. Keeping RAM refreshed so it retains its data.
2. Keeping CPUs asleep but ready to wake up when an external event happens.
3. Running the various radios: Cell, Wi-Fi, Bluetooth, etc.
4. Maintaining an active network connection to wake up when important events happen, such as receiving an instant message that should notify the user.
5. Apps doing work users may care about: syncing email (and possibly notifying of a new message arriving), updating current weather information for them to see next time they check their device, syncing news to show them current headlines next time they look, etc.

The more power consumed while not in use, the more the user's experience degrades due to there being less time she can actually use her device qon a single charge during the day. Most of the above items simply must be done to keep the

device functional, but the last point is more complicated: these are not necessary, and though they do create a better experience individually, this comes at the price of a worse overall battery life experience.

Consider a single app developer whose app lets you see news stories. It is important for people using the app to see the current news, possibly even for them to get notifications about recent news of interest, so the developer decides to refresh its news from the network even when the app is not directly in use. Of course the developer understands that just keeping the app running all of the time to constantly retrieve news is not good for the user, so a decision is made to do this only, say, twice an hour, to avoid draining the battery.

An app like this, doing some background work twice an hour, probably by itself has a good balance between experience in the app vs. overall battery life. However, now take 20 apps making this same trade-off and install them on a device: there is something wanting to do work in the background every 1.5 minutes! This will notably consume the device's limited available battery power, and thus how much it can be used during the day.

This problem is an illustration of the economic science concept of the *tragedy of the commons*. This is a situation where, when there are individual users of a shared resource, making their own individual rational decision about how to use that resource, together those decisions can result in over-consumption of the resource that results in harm to all of them beyond the individual benefits any each user gains. None of the individuals need be malicious in any way for this to happen. The original example of the tragedy of the commons is a public pasture for grazing sheep. It is in the interest of each farmer to have as many sheep as possible, but this may result in so many sheep that the pasture is overgrazed and all the sheep starve.

Android's approach of providing generic flexible building blocks for apps is a recipe for these kinds of tragedy of the commons issues. This design was important early on for Android to allow significant innovation on top of the platform in ways it could not anticipate. However, it also relies significantly on applications making good *global* decisions about their behavior. In particular, when an application asks to start a service, the platform must generally respect that (as much as it can) and allow the service to run, doing whatever it decides to do, until the app says it is done.

The most obvious problem this allows, however, is for apps that are poorly designed or buggy to rapidly drain the battery: starting a service for a long time, sitting there holding a wake lock keeping the device running, doing significant work on the CPU that uses power. Android 2.3 included the first major step in addressing background app battery use, shown in Fig. 10-83, which presents to the user how much the battery has drained and approximations for how much apps and other things on the device are responsible for that drain.

Viewing OS resource management as an economic/social problem, we have now seen two general strategies for addressing them. Tying foreground services to

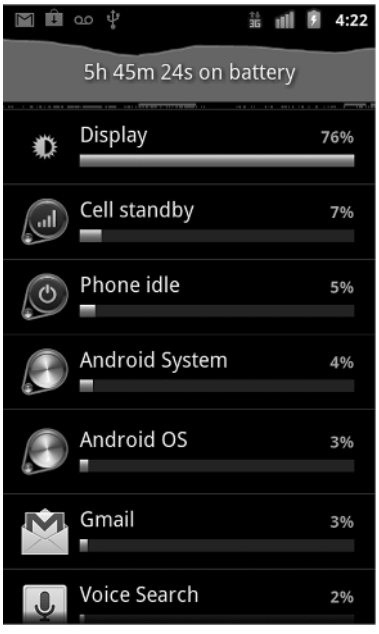

Figure 10-83. Android's early battery use screen.

notifications is an example of creating incentives that achieve the desired outcome: in this case a strong disincentive to abuse foreground services, because the associated notification will annoy people and give them a negative impression of the app. The battery usage display is an example of creating accountability: making visible the things applications are doing that can have significant impact on the device, so they can be held accountable for bad behavior and allow the user to take action based on that.

Neither of these approaches helps address the tragedy of the commons problem, where many reasonably behaving apps together consume too much power. It is difficult to find incentives that would significantly change the decisions those apps make (or even clearly say what the right decision is for each app), and accountability from battery usage data would simply show a large number of apps each individually using a small amount of the overall power. This was not initially a significant issue for Android, but as time went on, and devices had increasing numbers of apps installed on them, and those apps grew increasing amounts of functionality, it needed to be addressed.

Android 5.0 made the first major step at addressing cross-app power consumption problems with the introduction of the *JobScheduler* API. This provides a new specialized kind of service, one the app does not explicitly start or bind to, but instead tells the platform information about when it should run, such as whether it

needs network access, how frequently it should run, etc. Android then decides when to run the service and for how long.

JobScheduler gave Android the ability to look at the background work desires across all of the applications on a device and make scheduling decisions to balance how much work each app can do vs. their overall impact on battery life. For example, if Android determines that a particular app has not been used recently, it can significantly reduce how much work that app can do in the background in favor of other apps that are apparently more important to the user.

For JobScheduler to actually have an impact, however, apps need to use it; yet on its own, there is little incentive for them to do so. It did not replace the underlying flexible service mechanism, which apps were already using, were often easier to use (in a more simplistic way than jobs), and allowed them total flexibility to do the scheduling they wanted. Further changes were needed to change this situation.

Android 6.0 took the next step in taking more control over background execution by introducing "doze mode." The idea here was to identify one specific use case where battery life is a clear problem, and thus where strong restrictions could be applied by the platform to get significant gains. The target use case here was tablets that are not used for days: if the user leaves their tablet sitting on a shelf for a day, it is a terrible experience to come back to it with the battery empty. There is also no reason for users to have that experience, because they generally do not care about the tablet doing much of anything in the background during that time.

Doze addressed these long periods by defining it as a clear state the device can identify itself as being in, and stop all background work it can. Going into this state happens when the screen has been off for more than an hour and the device has not been moving. At that point, numerous restrictions are placed on the device: apps do not have network access and cannot hold wakelocks (so even if they have a running service they cannot keep the device consuming power), as well as other limitations such as turning off Wi-Fi and Bluetooth scans, limiting and throttling alarms, etc.

A device comes out of doze when the screen is turned back on or it is moved significantly (and thus needs to do scans and other things to collect new location-related information). The latter is accomplished by a special feature in the sensor system called a "significant motion detector" that allows the main CPU to go to sleep but wake up if the detector triggers.

While in doze, there is still a need to keep some limited background work happening. For example, an incoming instant message should still trigger a notification on the device, and important background operations should still be able to run for some amount. These needs are addressed through two mechanisms:

1. Android always maintains a connection to a server that tells it about important real-time events it should deal with, such as incoming instant messages or changes in calendar events. These are normally not delivered during doze, but a special high priority message allows these critical events to briefly wake up the device and handle them without impacting the overall doze state.

2. While in doze, the system will go into short *maintenance windows*, shown in Fig. 10-84, where most doze restrictions are released; this allows some continued operation of things like background syncing of email, refreshing news, etc.

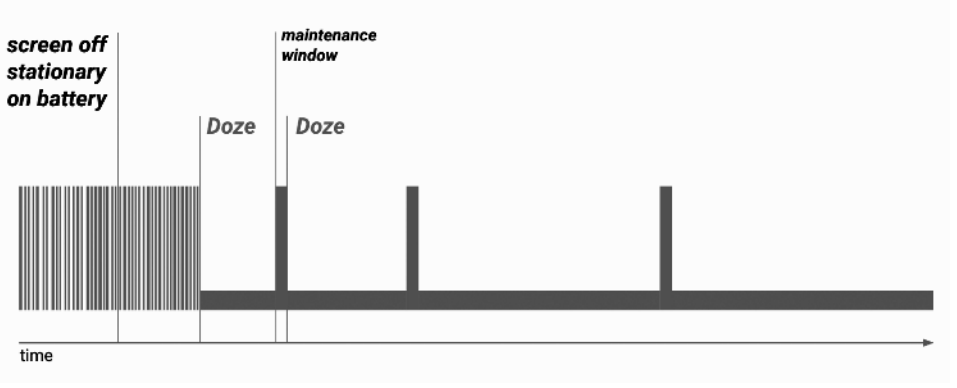

Figure 10-84. Doze and maintenance windows.

Apps can coordinate their work with doze maintenance windows through the previously mentioned JobScheduler. During doze, jobs are not scheduled, and the maintenance window is primarily a period when important pending jobs will be run by the system. This is the first significant incentive Android introduced for apps to switch from raw services to jobs, since services cannot as easily coordinate the work they are doing with the inability to access the network or hold wake locks during doze.

Android 7.0 created a new doze mode called "doze light." This applies many of the background restriction benefits of doze to most cases when a device's screen is off, even when it is being moved around. After the screen is off for a short period (around 15 minutes), doze light will kick in and apply the same network and wake lock restrictions as regular doze. Maintenance windows also exist in this mode, although they are much briefer in both duration and period between them. Since the device is allowed to be moving around in this mode, lower-level work like Wi-Fi and Bluetooth scans must be allowed to run.

Unfortunately, doze did not create sufficient incentives for apps to switch to JobScheduler (or at least to do this quickly), so Android 8.0 took a stronger approach with the creation of *background execution restrictions*. This applied a hard rule that most applications simply could no longer freely use plain services for background work, and now had to use JobScheduler. (At the same time, a new more explicit exception was created for purely foreground services in order to continue supporting their use cases.)

There is a mechanism for apps to remove background restrictions from themselves, through the explicit user interface mechanism previously discussed on the

topic of permissions. This requires the user to make a deliberate decision to give up their device's battery life to the app, which is a fairly high bar for most users; the result was sufficient pressure to drive most apps to finally move to JobScheduler instead.

Android 10 included a new restriction on activity launches. Prior to this release, an application in the background could freely launch an activity into the foreground. A number of use cases that needed this capability (such as incoming calls and alarm clocks) now had other facilities for getting the user's attention, and this capability was increasingly abused by malware. Disallowing background launches was done primarily to address the malware issue, but also closed a door apps had to get away from Android's background execution control: if they happened to be able to run a little bit in the background (such as receiving a broadcast), they could launch one of their activities to bring their app back to the foreground and escape any current background restrictions.

The changes up to Android 8, and to some degree the activity launch restrictions in Android 10, put the system in a much better position to manage the battery and ensure that users have a good experience. The state of things looked good for a few years, until a new issue started appearing: foreground services.

Recall that a foreground service is a special state for a service, marking it as important to the user. This state means that background restrictions and doze can not be applied to its app, for example, a foreground service being used to play music needs to run indefinitely, be able to keep the device awake, and have network access in case it is streaming audio from a server.

When background execution restrictions were implemented, an additional special carve-out needed to be created for foreground services. There are important cases where an app in the background will need to start a foreground service, such as starting their music playback in response to a media button being pressed while the app is not in the foreground. This has the same result as launching activities in the background, allowing them to escape background execution restrictions.

At this point, the original incentive to use foreground services for their intended purpose (doing something the user directly cares about), by requiring a notification, had broken down. Two major changes caused this. First, the increasing restrictions on background execution removed the alternative developers had of just using a regular service. Second, changes to the notification system had made app abuse of notifications less of a problem for them: originally, if the user was unhappy with a notification, their only option was to turn off all of the app's notifications. This prevented the app from getting the user's attention anywhere, since it could no longer post any notifications. Recent changes In Android allowed users to have finer control over notifications, so they could easily just hide the one for the foreground service without impacting other notifications.

Android 12 finally took on this problem by restricting foreground services. Much like the restriction on launching activities, applications could no longer start foreground services whenever they wanted. Instead, foreground services could now

only be started when the app was in a state where it was considered okay to do so, such as any time the app itself was already in the foreground for another reason, or it was executing in response to something that could be related to a user intent (such as responding to the aforementioned media button event).

This leaves us at the state of background execution in Android, circa 2021. Android will, however, continue to evolve; not only to continue to optimize the battery life it can provide, but also as it has to address the changing behavior of its application ecosystem and expectations of its users.

10.9 SUMMARY

In this chapter we have looked at two examples in detail: Linux and Android, which built on top on Linux. Linux has been around now for a bit over 30 years and has grown from a hobby project by one person who wanted a production version of MINIX to a large and powerful system that powers most of the Internet. It is also the most successful open-source project in history.

We started with a brief overview of the user interface and the shell, with some examples of what you can do on the command line. Then we took a brief look at some of the standard UNIX programs that are available in Linux. Next we saw how Linux is structured in layers.

After that we moved onto the core of the Linux material, how it works inside. This included processes and threads, memory management, input/output, the file system, and, of course, security. For each of these we showed some of the system calls available and how they are implemented. Then we moved on to Android, which is layered on top of Linux. Linux itself is mostly used on desktops, notebooks, and servers but Android is aimed at mobile devices such as smartphones and tablets. This changes its goals and requirements considerably. For example, how long it takes to start a program is of only minor interest on notebooks, but it is crucial on mobile devices. Notebook users really don't mind if Word takes 3 seconds to start, but smartphone users would go crazy if hitting the app to make a phone call took 3 seconds to boot. This simple difference in goals has vast implications for the respective designs.

Another enormous difference between Linux and Android is that while Linux tries to avoid wasting energy, Android goes to great lengths to prevent draining the battery to fast. A smartphone whose battery lasted only 4 hours would not be a big hit.

After going over tge design goals and history of Android, we took a look at the system architecture. Then we studied the innards in detail, includng ART, binder IPC, how apps work, intents, and the Android process model. Next come the ever-important security, which has evolved over the years in Android as it has become more important. Finally we looked at background execution, which is quite different from how it is done on desktops and notebooks.

PROBLEMS

1. The first version of UNIX was written in assembly code. Explain how writing UNIX in C made it easier to port it to new machines.

2. What is a portable C compiler? How does it simplify portability of UNIX?

3. The POSIX interface defines a set of library procedures. Explain why POSIX standardizes library procedures instead of the system-call interface.

4. Linux depends on gcc compiler to be ported to new architectures. Describe one advantage and one disadvantage of this dependency.

5. When the kernel catches a system call, how does it know which system call it is supposed to carry out?

6. What is the difference, if any between these two Linux command lines? Think about all possible cases.

```
cat f1 f2 f3 | grep "day" | head -500

cat f1 f2 f3 | grep "day" >tmp; head -500 tmp; rm tmp
```

7. What does the following Linux shell pipeline do?

```
grep rt xyz | wc –l
```

8. When the Linux shell starts up a process, it puts copies of its environment variables, such as *HOME*, on the process' stack, so the process can find out what its home directory is. If this process should later fork, will the child automatically get these variables, too?

9. About how long does it take a traditional UNIX system to fork off a child process under the following conditions: text size = 100 KB, data size = 20 KB, stack size = 10 KB, task structure = 1 KB, user structure = 5 KB. The kernel trap and return takes 1 msec, and the machine can copy one 32-bit word every 50 nsec. Text segments are shared, but data and stack segments are not.

10. As multimegabyte programs became more common, the time spent executing the fork system call and copying the data and stack segments of the calling process grew proportionally. When fork is executed in Linux, the parent's address space is not copied, as traditional fork semantics would dictate. How does Linux prevent the child from doing something that would completely change the fork semantics?

11. Why are negative arguments to nice reserved exclusively for the superuser?

12. A non-real-time Linux process has priority levels from 100 to 139. What is the default static priority and how is the nice value used to change this?

13. Does it make sense to take away a process' memory when it enters zombie state? Why or why not?

14. To what hardware concept is a signal closely related? Give two examples of how signals are used.

15. Why do you think the designers of Linux made it impossible for a process to send a signal to another process that is not in its process group?

16. There are a number of daemons running on most UNIX systems including Linux. Identify five daemons and provide a short description of each one. (*Hint*: Think about networking.)

17. When a new process is forked off, it must be assigned a unique integer as its PID. Is it sufficient to have a counter in the kernel that is incremented on each process creation, with the counter used as the new PID? Discuss your answer.

18. In every process' entry in the task structure, the PID of the parent is stored. Why?

19. The copy-on-write mechanism is used as an optimization in the fork system call, so that a copy of a page is created only when one of the processes (parent or child) tries to write on the page. Suppose a process *p1* forks processes *p2* and *p3* in quick succession. Explain how a page sharing may be handled in this case.

20. Two tasks A and B need to perform the same amount of work. However, task A has higher priority, and needs to be given more CPU time. Explain how will this be achieved in each of the Linux schedulers described in this chapter, the O(1) and the CFS scheduler.

21. Some UNIX systems are tickless, meaning they do not have periodic clock interrupts. Why is this done? Also, does ticklessness make sense on a computer (such as an embedded system) running only one process?

22. When booting Linux (or most other operating systems for that matter), the bootstrap loader in sector 0 of the disk first loads a boot program which then loads the operating system. Why is this extra step necessary? Surely it would be simpler to have the bootstrap loader in sector 0 just load the operating system directly.

23. A certain editor has 100 KB of program text, 30 KB of initialized data, and 50 KB of BSS. The initial stack is 10 KB. Suppose that three copies of this editor are started simultaneously. How much physical memory is needed (a) if shared text is used, and (b) if it is not?

24. Why are open-file-descriptor tables necessary in Linux?

25. In Linux, the data and stack segments are paged and swapped to a scratch copy kept on a special paging disk or partition, but the text segment uses the executable binary file instead. Why?

26. A DAX file system does not use a page cache. When is such a file system appropriate? Would you use a DAX file system with a hard disk? Why or why not?

27. Describe a way to use mmap and signals to construct an interprocess-communication mechanism.

28. A file is mapped in using the following mmap system call:

```
mmap(65536, 32768, READ, FLAGS, fd, 0)
```

Pages are 8 KB. Which byte in the file is accessed by reading a byte at memory address 72,000?

29. After the system call of the previous problem has been executed, the call

```
munmap(65536, 8192)
```

is carried out. Does it succeed? If so, which bytes of the file remain mapped? If not, why does it fail?

30. Can a page fault ever lead to the faulting process being terminated? If so, give an example. If not, why not?

31. Is it possible that with the buddy system of memory management it ever occurs that two adjacent blocks of free memory of the same size coexist without being merged into one block? If so, explain how. If not, show that it is impossible.

32. It is stated in the text that a paging partition will perform better than a paging file. Why is this so?

33. Give two examples of the advantages of relative path names over absolute ones.

34. The following locking calls are made by a collection of processes. For each call, tell what happens. If a process fails to get a lock, it blocks.

(a) *A* wants a shared lock on bytes 0 through 10.
(b) *B* wants an exclusive lock on bytes 20 through 30.
(c) *C* wants a shared lock on bytes 8 through 40.
(d) *A* wants a shared lock on bytes 25 through 35.
(e) *B* wants an exclusive lock on byte 8.

35. Consider the locked file of Fig. 10-26(c). Suppose that a process tries to lock bytes 10 and 11 and blocks. Then, before *C* releases its lock, yet another process tries to lock bytes 10 and 11, and also blocks. What kinds of problems are introduced into the semantics by this situation? Propose and defend two solutions.

36. Explain under what situations a process may request a shared lock or an exclusive lock. What problem may a process requesting an exclusive lock suffer from?

37. Suppose that an lseek system call seeks to a negative offset in a file. Given two possible ways of dealing with it.

38. If a Linux file has protection mode 755 (octal), what can the owner, the owner's group, and everyone else do to the file?

39. Some tape drives have numbered blocks and the ability to overwrite a particular block in place without disturbing the blocks in front of or behind it. Could such a device hold a mounted Linux file system?

40. In Fig. 10-24, both Aron and Nathan have access to the file *x* in their respective directories after linking. Is this access completely symmetrical in the sense that anything one of them can do with it the other one can, too?

41. As we have seen, absolute path names are looked up starting at the root directory and relative path names are looked up starting at the working directory. Suggest an efficient way to implement both kinds of searches.

42. When the file */usr/ast/work/f* is opened, several disk accesses are needed to read i-node and directory blocks. Calculate the number of disk accesses required under the assumption that the i-node for the root directory is always in memory, and all directories are one block long.

43. A Linux i-node has 12 disk addresses for data blocks, as well as the addresses of single, double, and triple indirect blocks. If each of these holds 256 disk addresses, what is the size of the largest file that can be handled, assuming that a disk block is 1 KB?

44. When an i-node is read in from the disk during the process of opening a file, it is put into an i-node table in memory. This table has some fields that are not present on the disk. One of them is a counter that keeps track of the number of times the i-node has been opened. Why is this field needed?

45. On multi-CPU platforms, Linux maintains a *runqueue* for each CPU. Is this a good idea? Explain your answer?

46. The concept of loadable modules is useful in that new device drivers may be loaded in the kernel while the system is running. Provide two disadvantages of this concept.

47. The kernel worker threads can be awakened periodically to write back to disk very old pages—older than 30 sec. Why is this necessary?

48. After a system crash and reboot, a recovery program is usually run. Suppose this program discovers that the link count in a disk i-node is 2, but only one directory entry references the i-node. Can it fix the problem, and if so, how?

49. Based on the information presented in this chapter, if a Linux ext2 file system were to be put on a 1.44-MB floppy disk, what is the maximum amount of user file data that could be stored on the disk? Assume that disk blocks are 1 KB.

50. In view of all the trouble that students can cause if they get to be superuser, why does this concept exist in the first place?

51. A professor shares files with his students by placing them in a publicly accessible directory on the Computer Science department's Linux system. One day he realizes that a file placed there the previous day was left world-writable. He changes the permissions and verifies that the file is identical to his master copy. The next day he finds that the file has been changed. How could this have happened and how could it have been prevented?

52. Linux has a system call fsuid. Unlike setuid, which grants the user all the rights of the effective id associated with a running program fsuid grants the user who is running the program special rights only with respect to access to files. Why is this feature useful?

53. On a Linux system, go to */proc/####* directory, where #### is a decimal number corresponding to a process currently running in the system. Answer the following along with an explanation:
(a) What is the size of most of the files in this directory?
(b) What are the time and date settings of most of the files?
(c) What type of access right is provided to the users for accessing the files?

54. If you are writing an Android activity to display a Web page in a browser, how would you implement its activity-state saving to minimize the amount of saved state without losing anything important?

55. If you are writing networking code on Android that uses a socket to download a file, what should you consider doing that is different than on a standard Linux system?

56. If you are designing something like Android's *zygote* process for a system that will have multiple threads running in each process forked from it, would you prefer to start those threads in zygote or after the fork?

57. Imagine you use Android's Binder IPC to send an object to another process. You later receive an object from a call into your process, and find that what you have received is the same object as previously sent. What can you assume or not assume about the caller in your process?

58. Consider an Android system that, immediately after starting, follows these steps:

1. The home (or launcher) application is started.
2. The email application starts syncing its mailbox in the background.
3. The user launches a camera application.
4. The user launches a Web browser application.

The Web page the user is now viewing in the browser application requires increasingly more RAM, until it needs everything it can get. What happens?

59. Consider the following Binder IPC scenario. Process *P1* has a Binder object implementing interface *I1*, Process *P2* has a Binder object implementing interface *I2*, Process *P3* has a Binder object implementing interface *I3*. Process *P1* creates a new Binder object with interface Ie; *P1* calls *I2* to send Ie to *P2*, then *P2* calls *I3* to send that Ie to *P3*, then *P3* calls *I1* to send that Ie to *P1*. *P1* now takes the Ie it received from *P3* and calls a method on it. What happens and why?

60. We have the following user journey through the Android system. Each application has one process associated with it.

1. Launch a "media player" application, and start playing music. The media player starts a foreground service to play the music.
2. The "media player" while playing music uses a content provider in another "audio server" app to retrieve the audio data it is playing.
3. Now return home, and start a "messaging" app.
4. In the "messaging" app, send a message to a friend, attaching an audio file. The messaging app is now using the content provider in the "audio server" app to retrieve the audio file.

5. While this is happening, a "email" app runs in the background to retrieve new messages from its server.

At this point with what we know, what is the importance category of the "media player," "audio server," "messaging," and "email" processes?

61. You have been told that Android has too many runtime permission prompts being shown to users, and you need to get rid of one of them. The current runtime prompts are (in the text shown to the user) "Contacts (access your contacts)," "Calendar (access your calendar)," "SMS (send and view SMS messages)," "Storage (access photos, media, and files)," "Location (access device's location)," "Phone (make and manage phone calls)," "Microphone (record audio)," "Camera (take pictures and record video)," and "Body sensors (access sensor data about your vital signs)." Which of these would you select to try to remove, and why?

62. You are starting to see a problem where users are doing many more explicit upload/download operations (such as sending large videos and recordings and downloading them), which apps correctly implement as foreground services. However, on devices that are more limited in RAM, these are conflicting with other foreground services like music playback, causing situations where the user's music is killed instead of uploads/downloads that would be a better choice. How might you solve this?

63. Write a minimal shell that allows simple commands to be started. It should also allow them to be started in the background.

64. Write a dumb terminal program to connect two Linux computers via the serial ports. Use the POSIX terminal management calls to configure the ports.

65. Write a client-server application which, on request, transfers a large file via sockets. Reimplement the same application using shared memory. Which version do you expect to perform better? Why? Conduct performance measurements with the code you have written and using different file sizes. What are your observations? What do you think happens inside the Linux kernel which results in this behavior?

66. Implement a basic user-level threads library to run on top of Linux. The library API should contain function calls like mythreads_init, mythreads_create, mythreads_join, mythreads_exit, mythreads_yield, mythreads_self, and perhaps a few others. Next, implement these synchronization variables to enable safe concurrent operations: mythreads_mutex_init, mythreads_mutex_lock, mythreads_mutex_unlock. Before starting, clearly define the API and specify the semantics of each of the calls. Next implement the user-level library with a simple, round-robin preemptive scheduler. You will also need to write one or more multithreaded applications, which use your library, in order to test it. Finally, replace the simple scheduling mechanism with another one which behaves like the Linux 2.6 O(1) scheduler described in this chapter. Compare the performance your application(s) receive when using each of the schedulers.

67. Write a shell script that displays some important system information such as what processes you are running, your home directory and current directory, processor type, current CPU utilization, etc.

68. Using assembly language and BIOS calls, write a program that boots itself from a USB drive on an x86 computer. The program should use BIOS calls to read the keyboard and echo the characters typed, just to demonstrate that it is running.