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, the system then establishes a connection between the sockets.

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 20 Mbps and outbound service at 2 Mbps, often under the name of **ADSL** (**Asymmetric Digital Subscriber Line**).

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 other related 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 idiosyncracies 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.

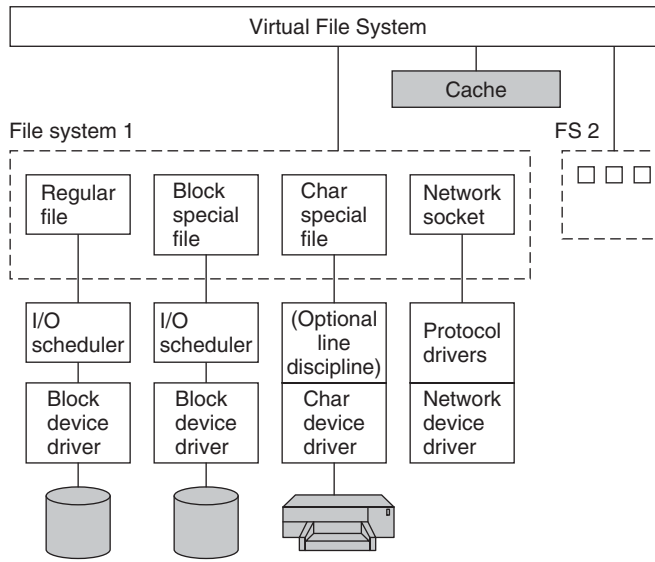

Figure 10-22. The Linux I/O system showing one file system in detail.

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 *pdflush* daemon 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.

In order to reduce the latency of repetitive disk-head movements, 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, also 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 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.

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 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/mos4/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 *mos4*, 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/mos4* 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/mos4/chap-10  /usr/ast/books/mos4/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). Fred and Lisa are working together on a project, and each of them needs access to the other's files. If Fred has */usr/fred* as his working directory, he can refer to the file *x* in Lisa's directory as */usr/lisa/x*. Alternatively, Fred can create a new entry in his directory, as shown in Fig. 10-24(b), after which he can use *x* to mean */usr/lisa/x*.

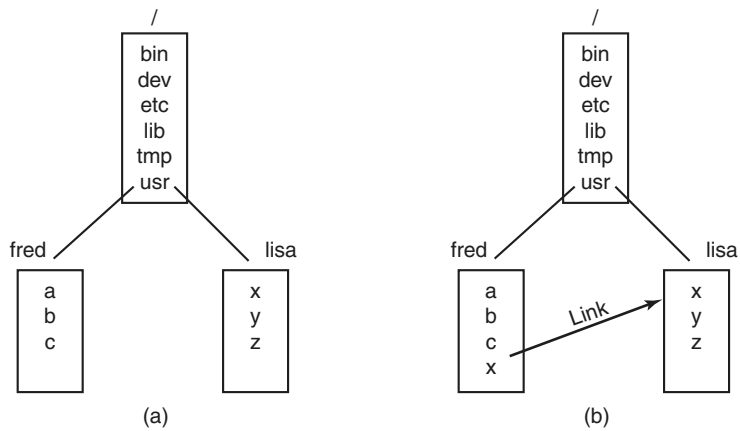

Figure 10-24. (a) Before linking. (b) After linking.

In the example just discussed, we suggested that before linking, the only way for Fred to refer to Lisa'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/fred*, another path to Lisa's file *x* is *../lisa/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. Even personal computers often have at least two disks—a hard disk and an optical (e.g., DVD) drive. 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 DVD, 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 8, which it inherited from MS-DOS in a century long ago.

The Linux solution is to allow one disk to be mounted in another disk's file tree. In our example, we could mount the DVD 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

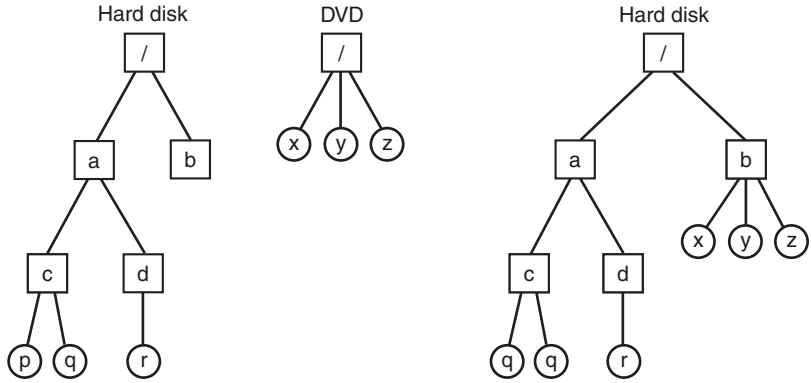

Figure 10-25. (a) Separate file systems. (b) After mounting.

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.

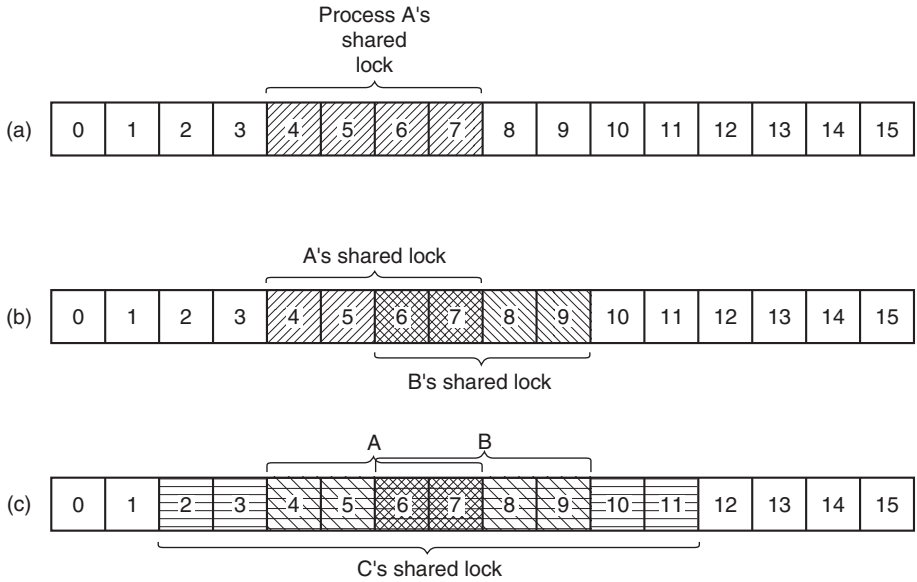

Figure 10-26. (a) A file with one lock. (b) Adding a second lock. (c) A third one.

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.) The parameters provide the name of the file and the protection mode. Thus

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

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. 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 seek 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	create, link
I-node	specific file	d_compare, d_delete
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 most popular 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 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 is still in 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.

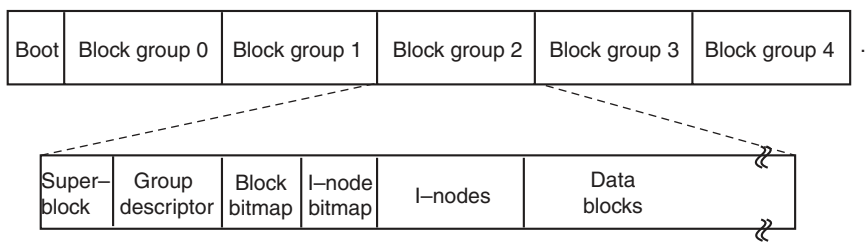

Figure 10-31. Disk layout of the Linux ext2 file system.

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

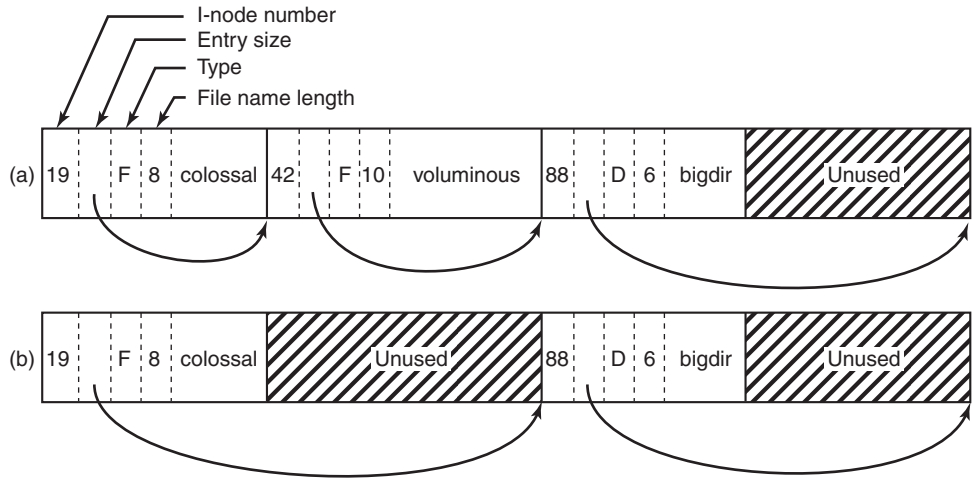

Figure 10-32. (a) A Linux directory with three files. (b) The same directory after the file *voluminous* has been removed.

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 see the same directory after the entry for *voluminous* has been removed. All the removeal 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 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 many 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, usually defaults to 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.

However, if an unrelated process opens the file, it gets its own open-file-description entry, with its own file position, which is precisely 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

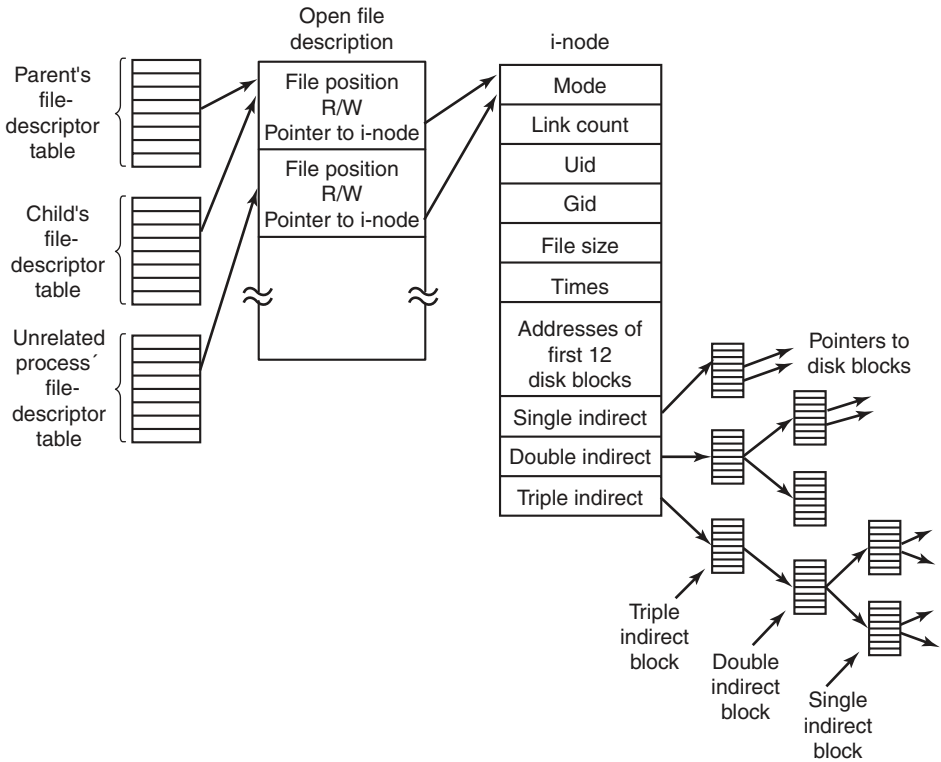

Figure 10-34. The relation between the file-descriptor table, the open-file-description-table, and the i-node table.

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. 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. Currently, the dominant NSF implementation is version 3, introduced in 1994. NSFv4 was 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.

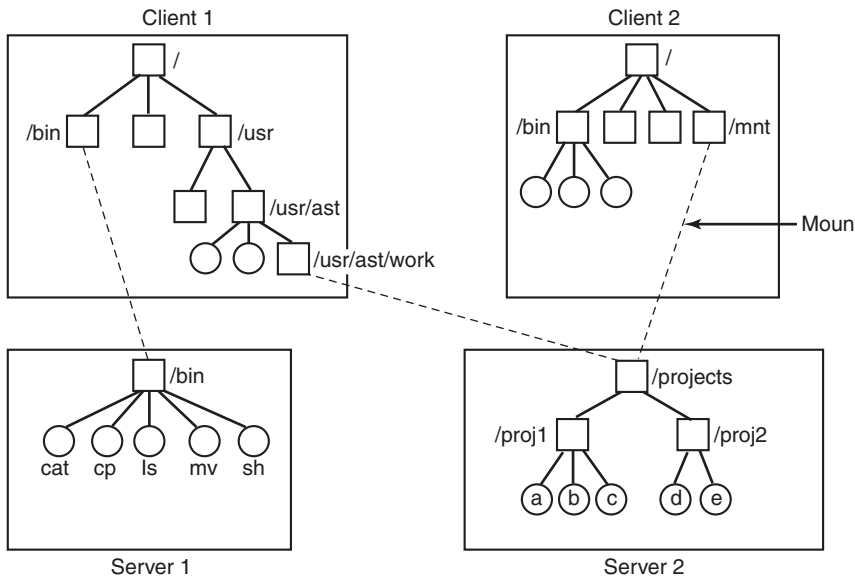

Figure 10-35. Examples of remote mounted file systems. Directories are shown as squares and files as circles.

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 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.

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. 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

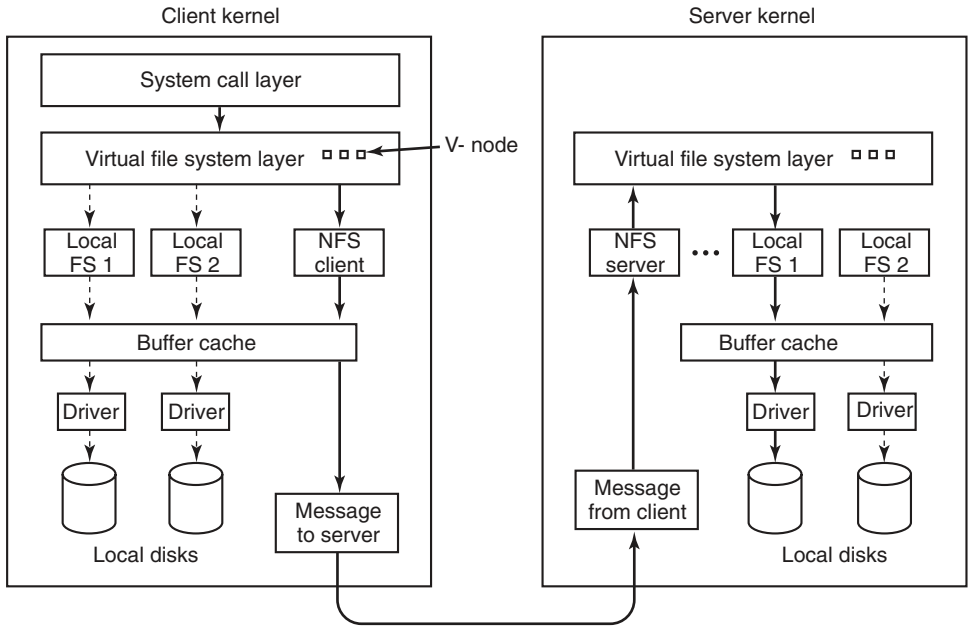

Figure 10-36. The NFS layer structure

file system and i-node are recorded because modern Linux systems can support multiple file systems (e.g., ext2fs, /proc, FAT, 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 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.

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.

In the five years since its introduction, Android has grown to be one of the most widely used smartphone operating systems. 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** (**GUI**) such as VOIP phones, smart watches, automotive dashboards, 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 **Android Open Source Project** (**AOSP**) 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 by itself 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 the 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 there being a GPS for performing mapping navigation) so the application is not made available on devices that lack those features.

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 interpreter (which runs the apps), system, 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:

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, significant 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.

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. The operating system 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, the more freedom they have to try out and install them.

5. Support typical mobile user interaction: spending 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 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 targeted and 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 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 Dalvik's Java language environment, which 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.

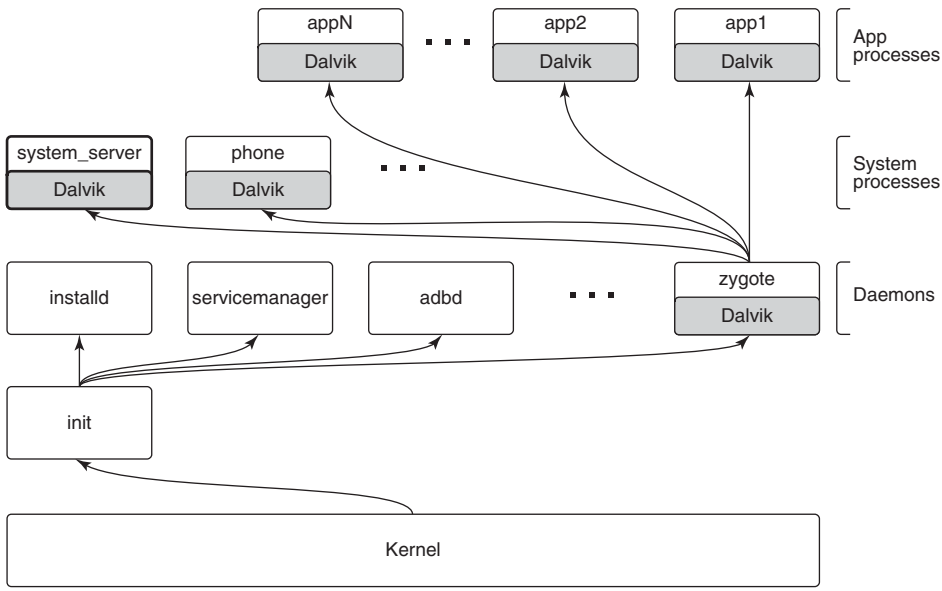

Figure 10-39. Android process hierarchy.

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 access, forking shell processes for them as needed.

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 must get a connection to the corresponding 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.

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, serveral 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.

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

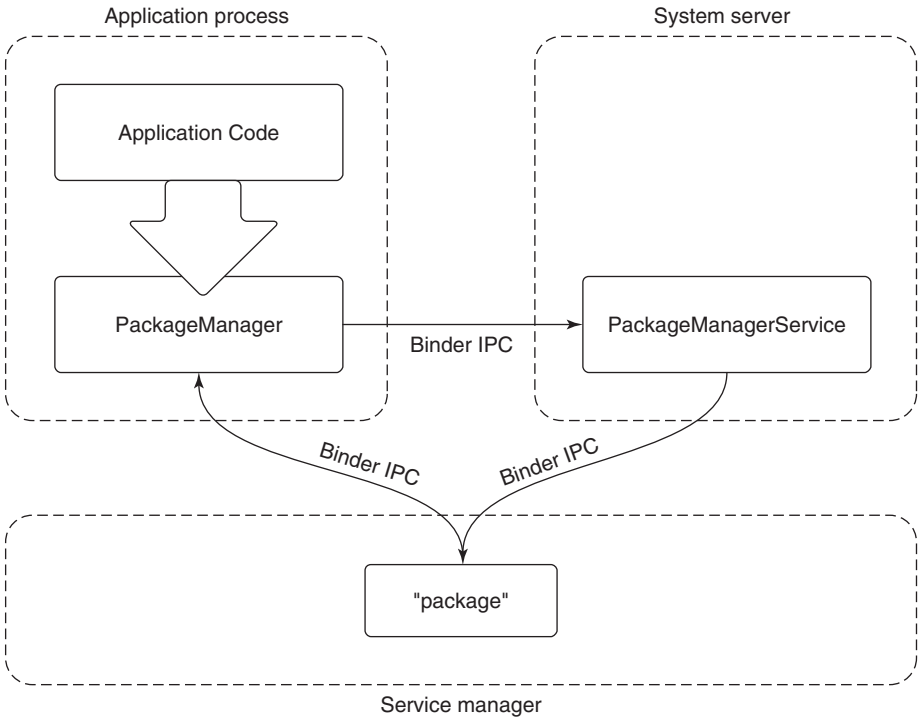

Figure 10-40. Publishing and interacting with system services.

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 button 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, 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 aquire 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.

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. Overcommit 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 introduces 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 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 out-of-memory killer uses 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.

10.8.6 Dalvik

Dalvik 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 Dalvik.

Android is not, however, a Java-language platform in the traditional sense. Java code in an Android application is provided in Dalvik's bytecode format, based around a register machine rather than Java's traditional stack-based bytecode. Dalvik's bytecode format allows for faster interpretation, while still supporting **JIT** (**Just-in-Time**) compilation. Dalvik bytecode 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 Dalvik's more compact bytecode representation. It is the Dalvik bytecode 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 for isolation between applications 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 Dalvik 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 Dalvik 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 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 Dalvik 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 previously. *Zygote* is responsible for bringing up and initializing Dalvik, to the point where it is ready to start running system or application code written in Java. All new Dalvik-based processes (system or application) are forked from *zygote*, allowing them to start execution with the environment already ready to go.

It is not just Dalvik 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, but there is no exec 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 Dalvik environment, though it is sharing all of the preloaded and initialed data with *zygote* through copy-on-write pages. All that now remains to have the new running process ready to go is to give it the correct identity (UID etc.), finish any initialization of Dalvik 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 Dalvik 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."

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

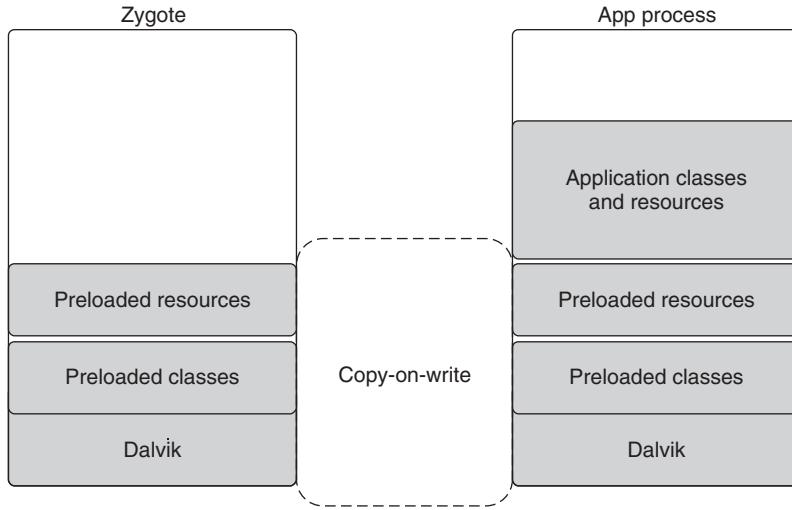

Figure 10-41. Creating a new *Dalvik* process from *zygote*.

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 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

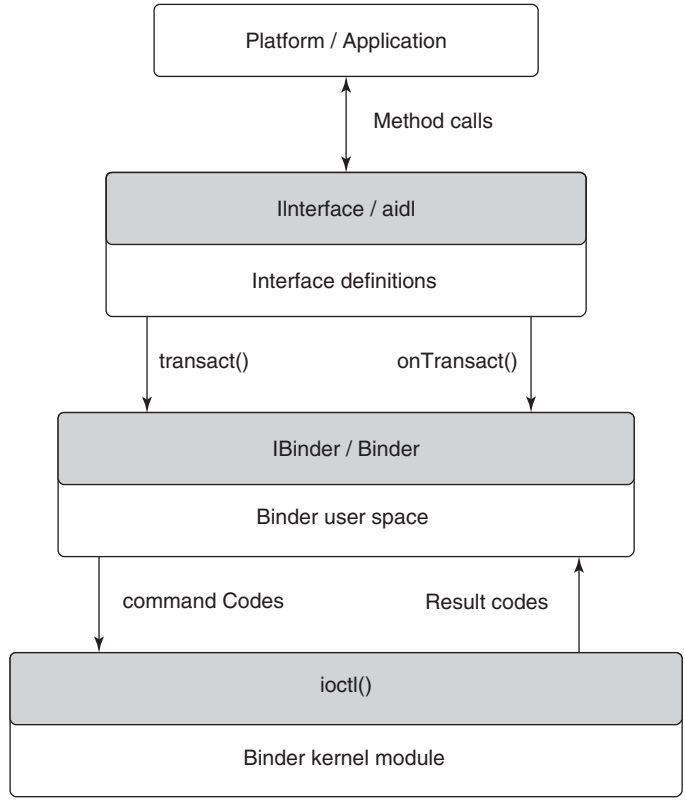

Figure 10-42. *Binder* IPC architecture.

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.

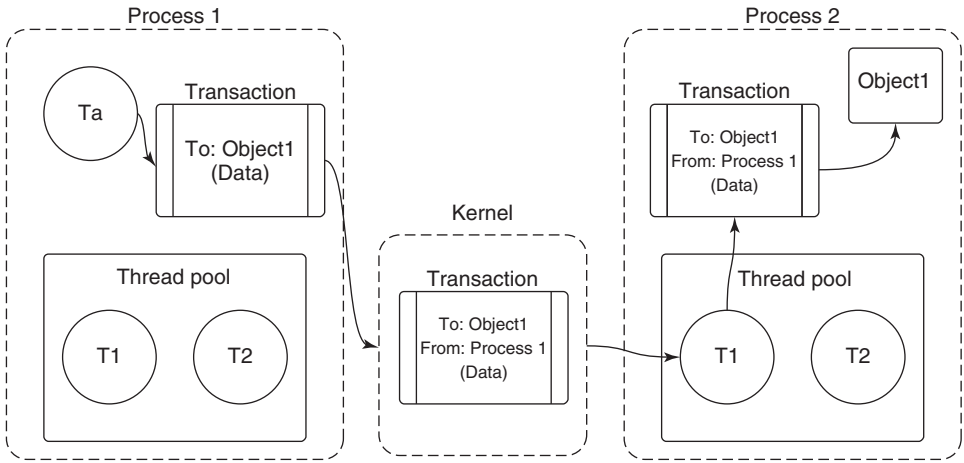

Figure 10-43. Basic *Binder* IPC transaction.

(For the discussion here, we are simplifying the 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.

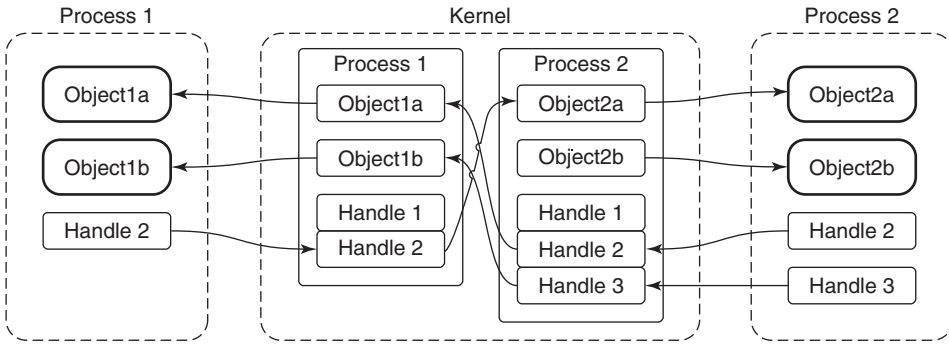

Figure 10-44. *Binder* cross-process object mapping.

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.

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

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.

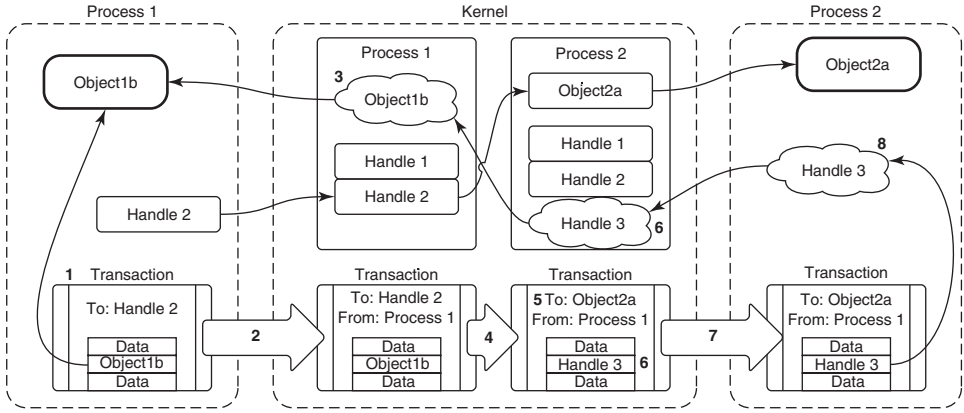

Figure 10-45. Transferring *Binder* objects between processes.

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 is 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.

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 is 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.

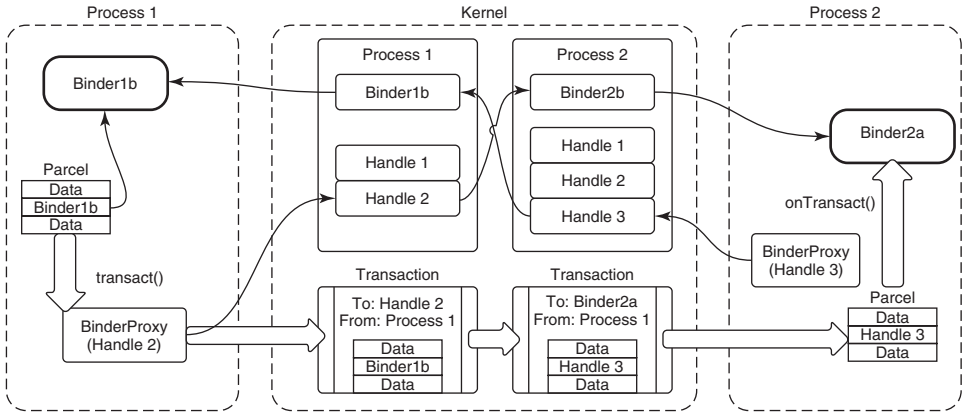

Figure 10-46. *Binder* user-space API.

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 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.

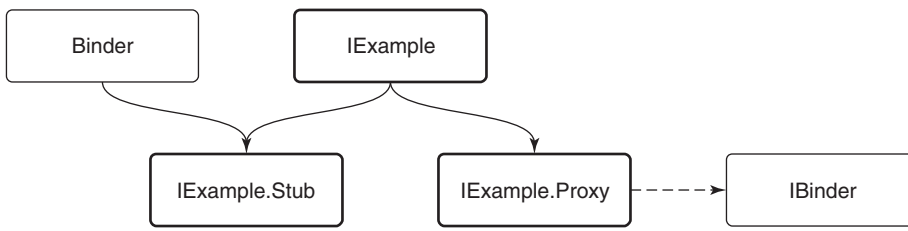

Figure 10-48. *Binder* interface inheritance hierarchy.

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 underlying *BinderProxy*.
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.

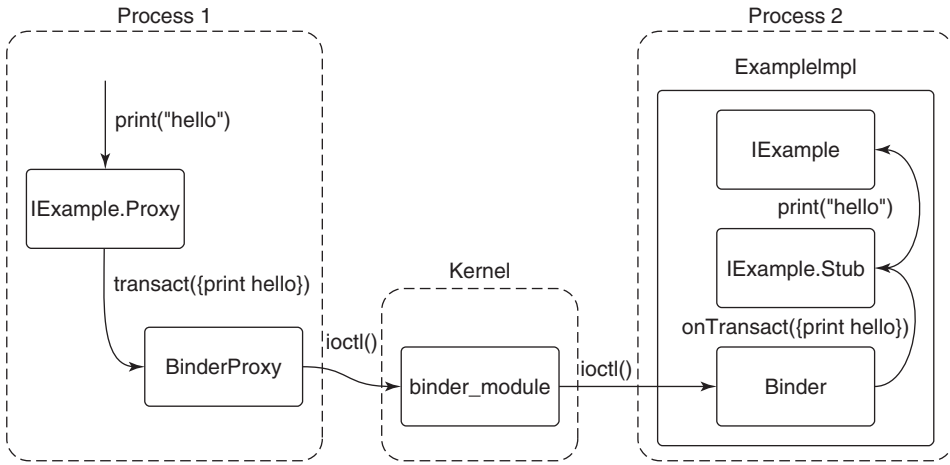

Figure 10-49. Full path of an *AIDL*-based *Binder* IPC.

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*.

10.8.8 Android Applications

Android provides an application model that is very different from the normal command-line environment in the Linux shell or even applications launched from a graphical user interface. 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.

```
package android.os

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

Figure 10-50. Basic service manager AIDL interface.

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:

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 Dalvik 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.

Android applications do not have a simple main entry point which 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.

```
<?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.

The **package manager** is the part of Android that keeps track of all application packages. 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 currently installed applications and retrieve relevant information about them. 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 (cleaning up everything associated with a previously installed 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: installing an application does not require running any application code, the top-level capabilities of the application can always be determined by looking at the 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 design 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.

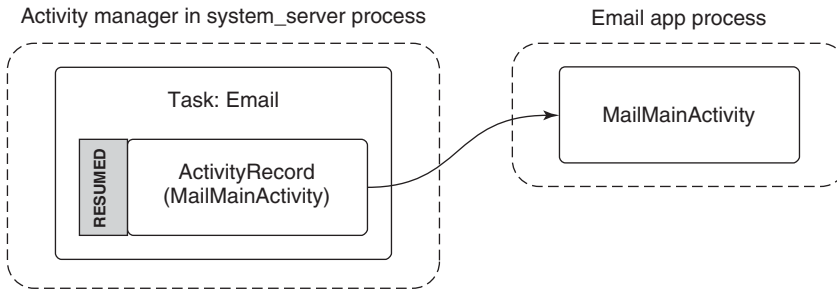

Figure 10-52. Starting an email application's main activity.

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*.

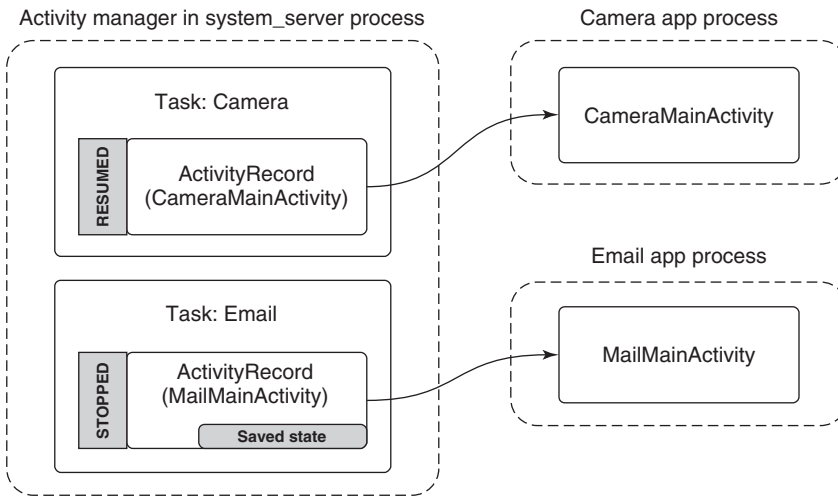

Figure 10-53. Starting the camera application after email.

When an activity is no longer in the foreground, the system asks it to "save its state." This involves the application creating a minimal amount of state information representing what the user currently sees, which it returns to the activity

manager and stores in the *system_server* process, in the *ActivityRecord* associated with that activity. The saved state for an activity is generally small, containing for example where you are scrolled in an email message, but not the message itself, which will be stored elsewhere by the application in its persistent storage.

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.

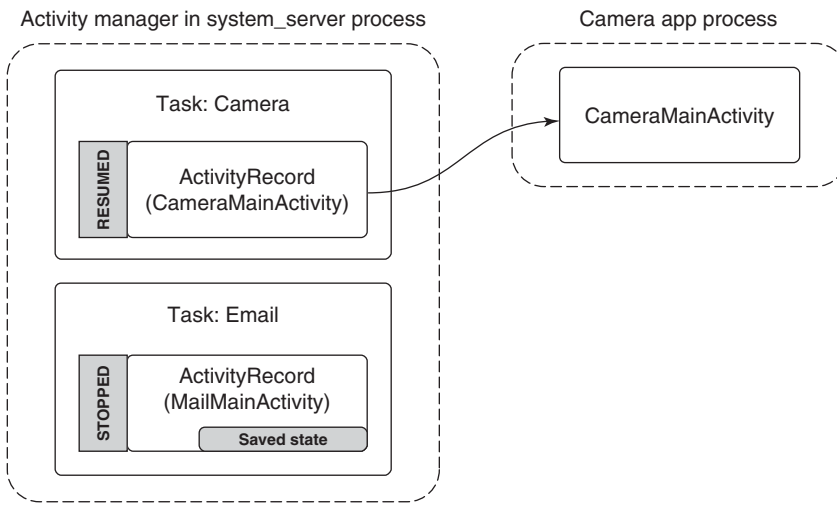

Figure 10-54. Removing the email process to reclaim RAM for the camera.

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 *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.

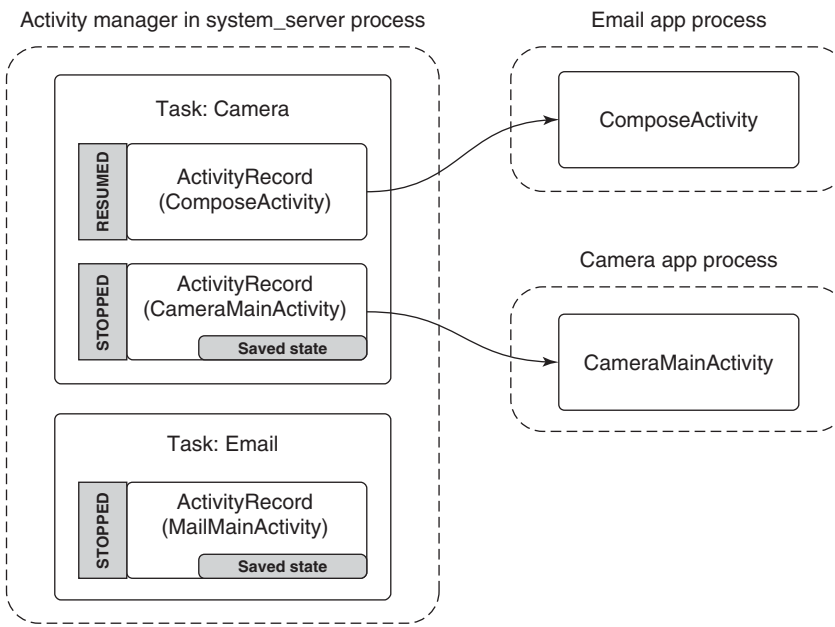

Figure 10-55. Sharing a camera picture through the email application.

Finally let us look at 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.

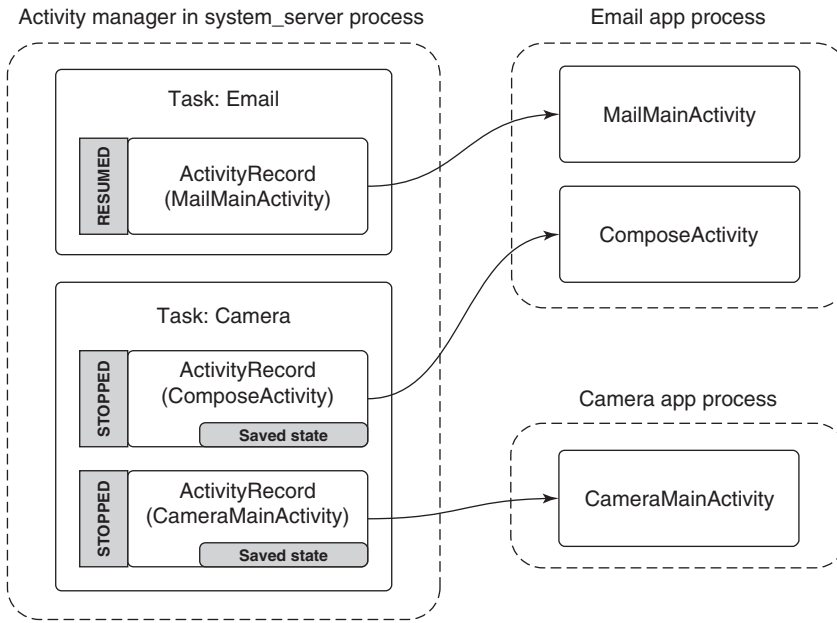

Figure 10-56. Returning to the email application.

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.

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.

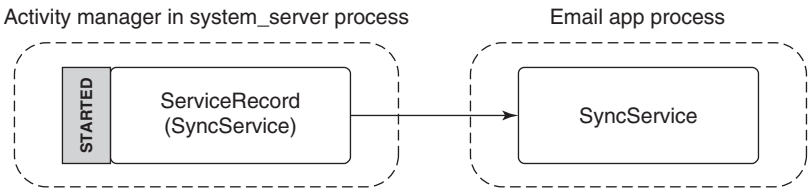

Figure 10-57. Starting an application service.

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:

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.

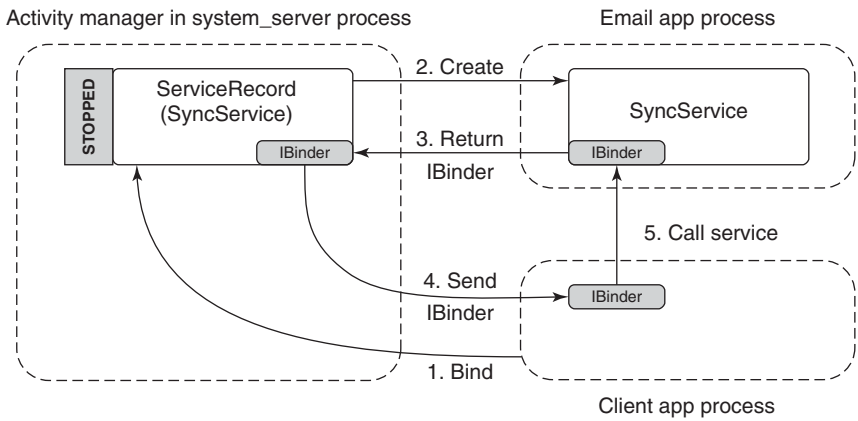

Figure 10-59. Binding to an application service.

Receivers

A **receiver** is the recipient of (typically external) events that happen, generally in the background and outside of normal user interaction. 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 *Broadcast-Record* 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.

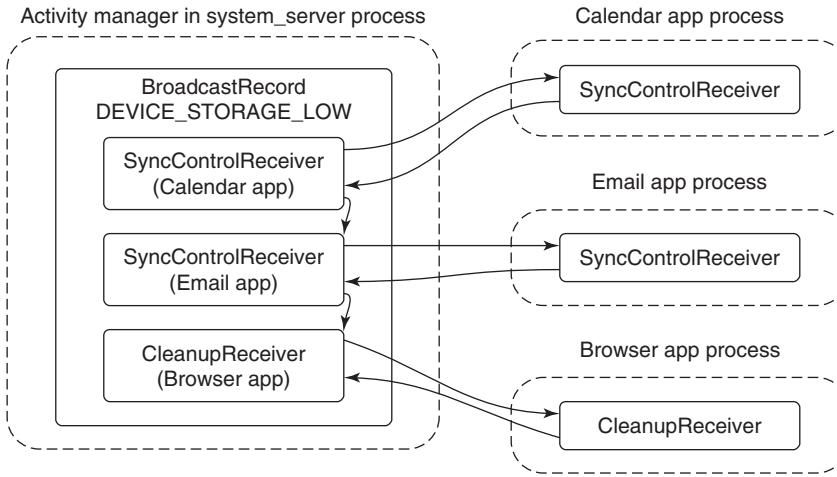

Figure 10-60. Sending a broadcast to application receivers.

Receivers only run as one-shot operations. 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.

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 which 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.

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 is 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.

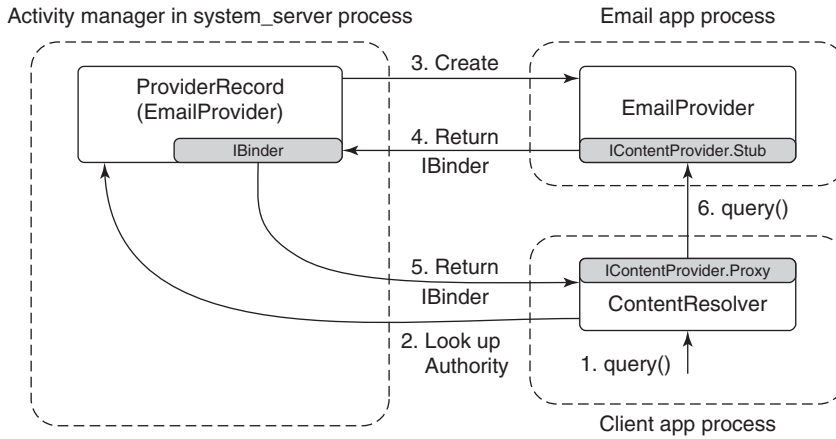

Figure 10-61. Interacting with a content provider.

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 activities 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 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, that have sweeping functionality, which will do a wide variety of things in their environment 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 warning the first time they are running an application, 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 in protecting users from themselves. All programs run with the power of the user and, if any of them misbehaves, it can do all the damage the user can do. 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!

Android attempts to address this 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 feature in Linux, but in a novel way. The result is better isolation.

10.8.11 Security

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 10-62 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 10-62. 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 free 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* provides to the package manager for it to create and manage the data directories of applications as needed.

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 prior to installing the application the user is informed of what it will be allowed to do based on them.

Figure 10-63 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 is returned to the application.

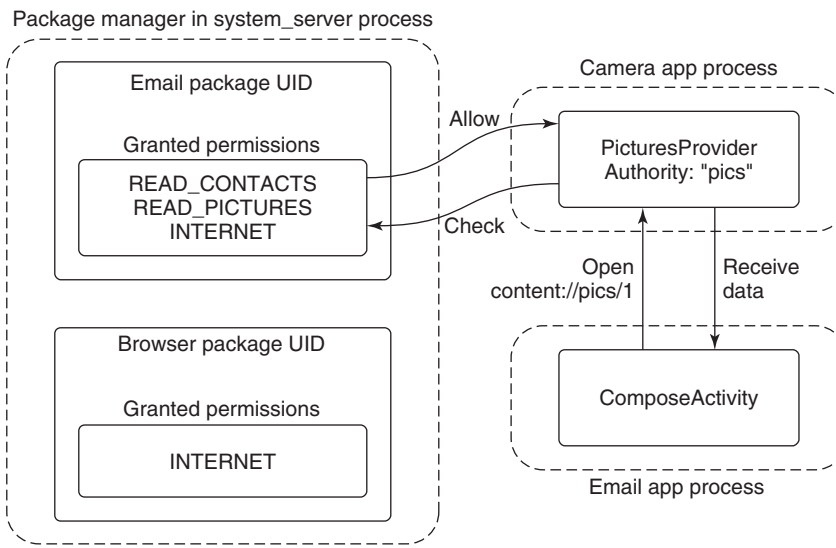

Figure 10-63. Requesting and using a permission.

Permissions are not tied to content providers; any IPC into the system may be protected by a permission through the system's 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 10-64 illustrates 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.

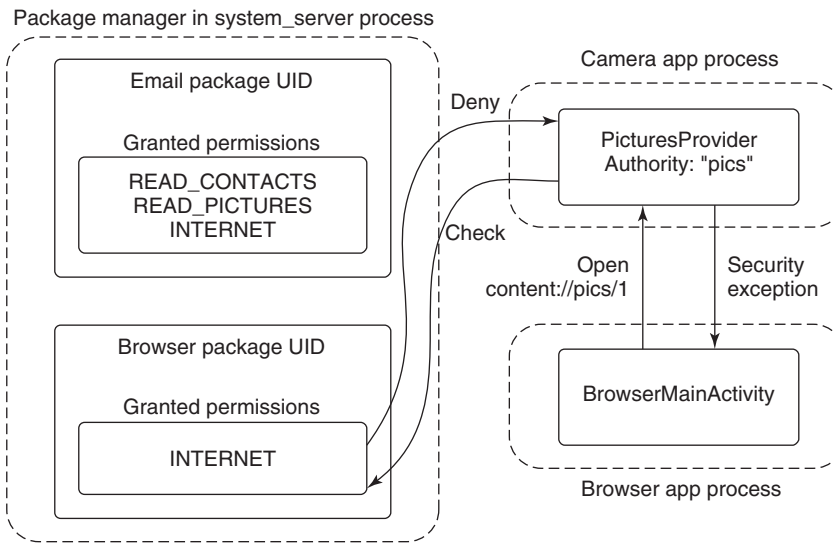

Figure 10-64. Accessing data without a permission.

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 your pictures, and no reason for an email application to have access to all of your 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 10-65 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 way. There is another intent action, android.intent.action.GET_CONTENT, which an application can use to ask the user to pick some data and return to it. 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 10-66 illustrates this new flow. It is almost identical to Fig. 10-65, 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 application. 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.

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.

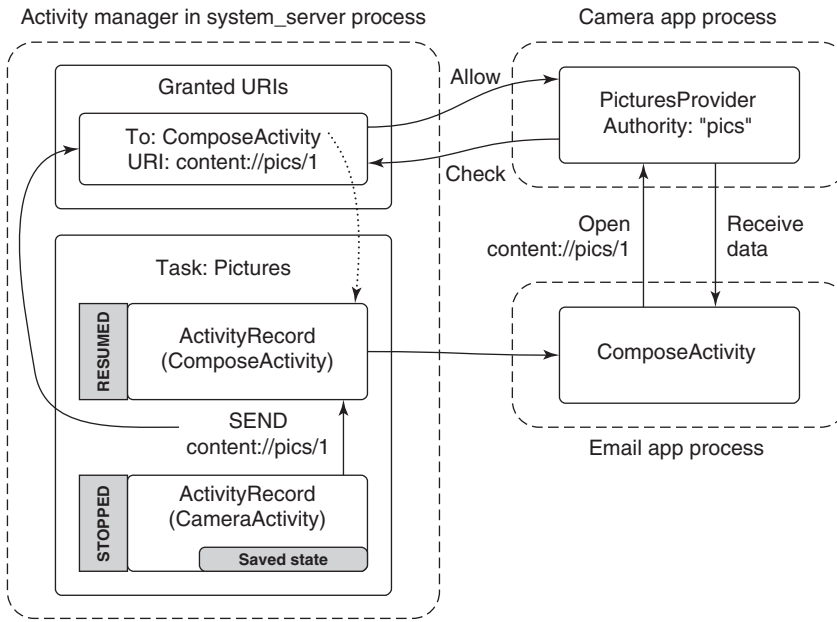

Figure 10-65. Sharing a picture using a content provider.

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.

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 the user wants to use a new input method after they have installed its 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.

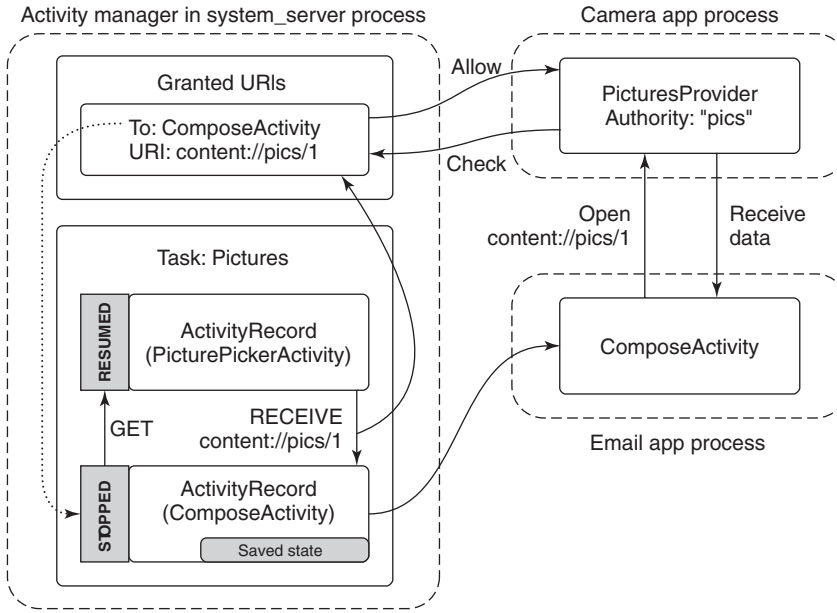

Figure 10-66. Adding a picture attachment using a content provider.

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.

10.8.12 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 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 UID it will run as, finally dropping root privileges and changing the process to the desired UID.

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. 10-67. The details of each step in the illustration are:

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 its UID appropriately for the application's sandbox, and finishes initialization of Dalvik 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.

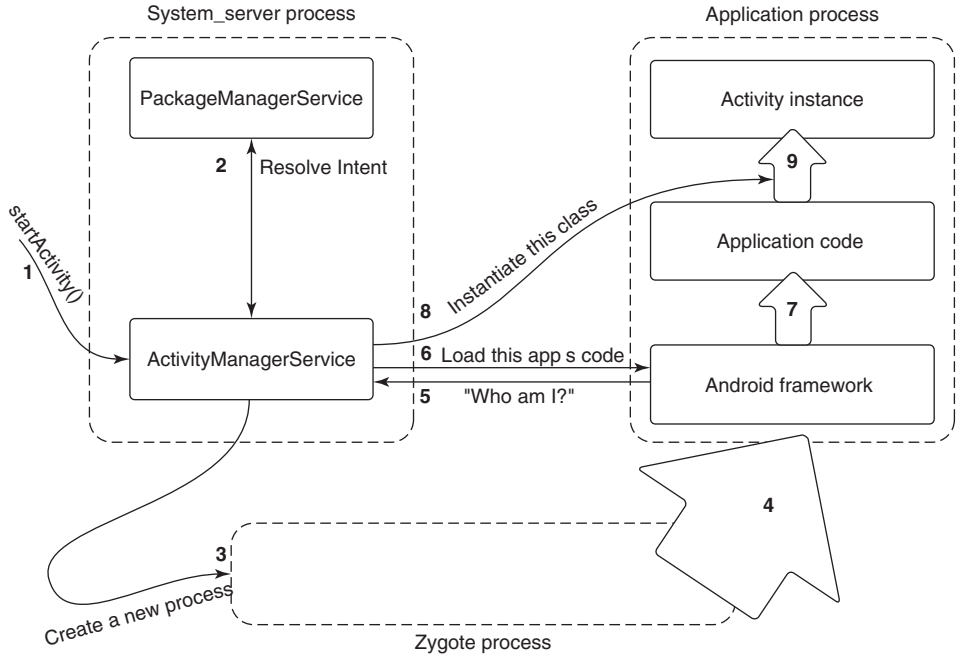

Figure 10-67. Steps in launching a new application process.

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 *oom_adj* as a strict ordering to determine which processes it should kill first. The activity manager is responsible for setting each process's *oom_adj* appropriately

based on the state of that process, by classifying them into major categories of use. Figure 10-68 shows the main categories, with the most important category first. The last column shows a typical *oom_adj* value that is assigned to processes of this type.

Category	Description	oom_adj
SYSTEM	The system and daemon processes	–16
PERSISTENT	Always-running application processes	–12
FOREGROUND	Currently interacting with user	0
VISIBLE	Visible to user	1
PERCEPTIBLE	Something the user is aware of	2
SERVICE	Running background services	3
HOME	The home/launcher process	4
CACHED	Processes not in use	5

Figure 10-68. 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 *oom_adj* 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 thrown out of active RAM. 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 at this point have a good overview of how individual Android processes are managed. There is a further complication to this, however: dependencies between processes.

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 *oom_adj* 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 10-69 illustrates a typical state processes can be in, taking into account dependencies between them. This example contains two dependencies, based on using a camera-content provider to add a picture attachment to an email as discussed in Fig. 10-66. First is the current foreground email application, which is making use of the camera application to load an attachment. This raises the camera process up to the same importance as the email app. Second is a similar situation, the 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.

Consider what happens if the state of Fig. 10-69 changes so that the email application is done loading the attachment, and no longer uses the camera content provider. Figure 10-70 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.

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.

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 10-69. 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 10-70. Process state after email stops using camera

10.9 SUMMARY

Linux began its life as an open-source, full-production UNIX clone, and is now used on machines ranging from smartphones and notebook computers to supercomputers. Three main interfaces to it exist: the shell, the C library, and the system calls themselves. In addition, a graphical user interface is often used to simplify user interaction with the system. The shell allows users to type commands for execution. These may be simple commands, pipelines, or more complex structures. Input and output may be redirected. The C library contains the system calls and also many enhanced calls, such as *printf* for writing formatted output to files. The actual system call interface is architecture dependent, and on x86 platforms consists of roughly 250 calls, each of which does what is needed and no more.

The key concepts in Linux include the process, the memory model, I/O, and the file system. Processes may fork off subprocesses, leading to a tree of processes.

Process management in Linux is different compared to other UNIX systems in that Linux views each execution entity—a single-threaded process, or each thread within a multithreaded process or the kernel—as a distinguishable task. A process, or a single task in general, is then represented via two key components, the task structure and the additional information describing the user address space. The former is always in memory, but the latter data can be paged in and out of memory. Process creation is done by duplicating the process task structure, and then setting the memory-image information to point to the parent's memory image. Actual copies of the memory-image pages are created only if sharing is not allowed and a memory modification is required. This mechanism is called copy on write. Scheduling is done using a weighted fair queueing algorithm that uses a red-black tree for the tasks' queue management.

The memory model consists of three segments per process: text, data, and stack. Memory management is done by paging. An in-memory map keeps track of the state of each page, and the page daemon uses a modified dual-hand clock algorithm to keep enough free pages around.

I/O devices are accessed using special files, each having a major device number and a minor device number. Block device I/O uses the main memory to cache disk blocks and reduce the number of disk accesses. Character I/O can be done in raw mode, or character streams can be modified via line disciplines. Networking devices are treated somewhat differently, by associating entire network protocol modules to process the network packets stream to and from the user process.

The file system is hierarchical with files and directories. All disks are mounted into a single directory tree starting at a unique root. Individual files can be linked into a directory from elsewhere in the file system. To use a file, it must be first opened, which yields a file descriptor for use in reading and writing the file. Internally, the file system uses three main tables: the file descriptor table, the open-file-description table, and the i-node table. The i-node table is the most important of these, containing all the administrative information about a file and the location of its blocks. Directories and devices are also represented as files, along with other special files.

Protection is based on controlling read, write, and execute access for the owner, group, and others. For directories, the execute bit means search permission.

Android is a platform for allowing apps to run on mobile devices. It is based on the Linux kernel, but consists of a large body of software on top of Linux, plus a small number of changes to the Linux kernel. Most of Android is written in Java. Apps are also written in Java, then translated to Java bytecode and then to Dalvik bytecode. Android apps communicate by a form of protected message passing called transactions. A special Linux kernel model called the *Binder* handles the IPC.

Android packages are self contained and have a manifest desccribing what is in the package. Packages contain activities, receivers, content providers, and intents. The Android security model is different from the Linux model and carefully sandboxes each app because all apps are regarded as untrustworthy.

PROBLEMS

1. Explain how writing UNIX in C made it easier to port it to new machines.
2. The POSIX interface defines a set of library procedures. Explain why POSIX standardizes library procedures instead of the system-call interface.
3. Linux depends on gcc compiler to be ported to new architectures. Describe one advantage and one disadvantage of this dependency.
4. A directory contains the following files:

aardvark	ferret	koala	porpoise	unicorn
bonefish	grunion	llama	quacker	vicuna
capybara	hyena	marmot	rabbit	weasel
dingo	ibex	nuthatch	seahorse	yak
emu	jellyfish	ostrich	tuna	zebu

   Which files will be listed by the command

```
ls [abc]*e*?
```

5. What does the following Linux shell pipeline do?

```
grep nd xyz | wc –l
```

6. Write a Linux pipeline that prints the eighth line of file *z* on standard output.
7. Why does Linux distinguish between standard output and standard error, when both default to the terminal?
8. A user at a terminal types the following commands:

```
a | b | c &
d | e | f &
```

   After the shell has processed them, how many new processes are running?

9. 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?
10. 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.
11. 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?

12. Why are negative arguments to nice reserved exclusively for the superuser?

13. 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?

14. Does it make sense to take away a process' memory when it enters zombie state? Why or why not?

15. To what hardware concept is a signal closely related? Give two examples of how signals are used.

16. 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?

17. A system call is usually implemented using a software interrupt (trap) instruction. Could an ordinary procedure call be used as well on the Pentium hardware? If so, under what conditions and how? If not, why not?

18. In general, do you think daemons have higher or lower priority than interactive processes? Why?

19. 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.

20. In every process' entry in the task structure, the PID of the parent is stored. Why?

21. 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.

22. What combination of the *sharing_flags* bits used by the Linux clone command corresponds to a conventional UNIX fork call? To creating a conventional UNIX thread?

23. 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. Expain how will this be achieved in each of the Linux schedulers described in this chapter, the O(1) and the CFS scheduler.

24. 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?

25. 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.

26. 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?

27. Why are open-file-descriptor tables necessary in Linux?

28. 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?

29. Describe a way to use mmap and signals to construct an interprocess-communication mechanism.

30. 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?

31. 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?

32. Can a page fault ever lead to the faulting process being terminated? If so, give an example. If not, why not?

33. 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.

34. It is stated in the text that a paging partition will perform better than a paging file. Why is this so?

35. Give two examples of the advantages of relative path names over absolute ones.

36. 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.

37. 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.

38. 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?

39. If a Linux file has protection mode 755 (octal), what can the owner, the owner's group, and everyone else do to the file?

40. 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?

41. In Fig. 10-24, both Fred and Lisa 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?

42. 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.

43. 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.

44. 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?

45. 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?

46. On multi-CPU platforms, Linux maintains a *runqueue* for each CPU. Is this a good idea? Explain your answer?

47. 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.

48. *Pdflush* threads can be awakened periodically to write back to disk very old pages—older than 30 sec. Why is this necessary?

49. 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?

50. Make an educated guess as to which Linux system call is the fastest.

51. Is it possible to unlink a file that has never been linked? What happens?

52. 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.

53. 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?

54. 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?

55. Linux supports a system call fsuid. Unlike setuid, which grants the user all the rights of the effective id associated with a program he is running, fsuid grants the user who is

running the program special rights only with respect to access to files. Why is this feature useful?

56. 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?

57. 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?

58. 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?

59. 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?

60. 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?

61. 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 inceasingly more RAM, until it needs everything it can get. What happens?

62. Write a minimal shell that allows simple commands to be started. It should also allow them to be started in the background.

63. Using assembly language and BIOS calls, write a program that boots itself from a floppy disk on a Pentium-class computer. The program should use BIOS calls to read the keyboard and echo the characters typed, just to demonstrate that it is running.

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.

11

CASE STUDY 2: WINDOWS 8

Windows is a modern operating system that runs on consumer PCs, laptops, tablets and phones as well as business desktop PCs and enterprise servers. Windows is also the operating system used in Microsoft's Xbox gaming system and Azure cloud computing infrastructure. The most recent version is Windows 8.1. In this chapter we will examine various aspects of Windows 8, starting with a brief history, then moving on to its architecture. After this we will look at processes, memory management, caching, I/O, the file system, power management, and finally, security.

11.1 HISTORY OF WINDOWS THROUGH WINDOWS 8.1

Microsoft's development of the Windows operating system for PC-based computers as well as servers can be divided into four eras: **MS–DOS**, **MS–DOS-based Windows**, **NT-based Windows**, and **Modern Windows**. Technically, each of these systems is substantially different from the others. Each was dominant during different decades in the history of the personal computer. Figure 11-1 shows the dates of the major Microsoft operating system releases for desktop computers. Below we will briefly sketch each of the eras shown in the table.

Year	MS–DOS	MS-DOS based Windows	NT-based Windows	Modern Windows	Notes
1981	1.0				Initial release for IBM PC
1983	2.0				Support for PC/XT
1984	3.0				Support for PC/AT
1990		3.0			Ten million copies in 2 years
1991	5.0				Added memory management
1992		3.1			Ran only on 286 and later
1993			NT 3.1		
1995	7.0	95			MS-DOS embedded in Win 95
1996			NT 4.0		
1998		98			
2000	8.0	Me	2000		Win Me was inferior to Win 98
2001			XP		Replaced Win 98
2006			Vista		Vista could not supplant XP
2009			7		Significantly improved upon Vista
2012				8	First Modern version
2013				8.1	Microsoft moved to rapid releases

Figure 11-1. Major releases in the history of Microsoft operating systems for desktop PCs.

11.1.1 1980s: MS-DOS

In the early 1980s IBM, at the time the biggest and most powerful computer company in the world, was developing a **personal computer** based the Intel 8088 microprocessor. Since the mid-1970s, Microsoft had become the leading provider of the BASIC programming language for 8-bit microcomputers based on the 8080 and Z-80. When IBM approached Microsoft about licensing BASIC for the new IBM PC, Microsoft readily agreed and suggested that IBM contact Digital Research to license its CP/M operating system, since Microsoft was not then in the operating system business. IBM did that, but the president of Digital Research, Gary Kildall, was too busy to meet with IBM. This was probably the worst blunder in all of business history, since had he licensed CP/M to IBM, Kildall would probably have become the richest man on the planet. Rebuffed by Kildall, IBM came back to Bill Gates, the cofounder of Microsoft, and asked for help again. Within a short time, Microsoft bought a CP/M clone from a local company, Seattle Computer Products, ported it to the IBM PC, and licensed it to IBM. It was then renamed **MS-DOS 1.0** (**MicroSoft Disk Operating System**) and shipped with the first IBM PC in 1981.

MS-DOS was a 16-bit real-mode, single-user, command-line-oriented operating system consisting of 8 KB of memory resident code. Over the next decade, both the PC and MS-DOS continued to evolve, adding more features and capabilities. By 1986, when IBM built the PC/AT based on the Intel 286, MS-DOS had grown to be 36 KB, but it continued to be a command-line-oriented, one-application-ata-time, operating system.

11.1.2 1990s: MS-DOS-based Windows

Inspired by the graphical user interface of a system developed by Doug Engelbart at Stanford Research Institute and later improved at Xerox PARC, and their commercial progeny, the Apple Lisa and the Apple Macintosh, Microsoft decided to give MS-DOS a graphical user interface that it called **Windows**. The first two versions of Windows (1985 and 1987) were not very successful, due in part to the limitations of the PC hardware available at the time. In 1990 Microsoft released **Windows 3.0** for the Intel 386, and sold over one million copies in six months.

Windows 3.0 was not a true operating system, but a graphical environment built on top of MS-DOS, which was still in control of the machine and the file system. All programs ran in the same address space and a bug in any one of them could bring the whole system to a frustrating halt.

In August 1995, **Windows 95** was released. It contained many of the features of a full-blown operating system, including virtual memory, process management, and multiprogramming, and introduced 32-bit programming interfaces. However, it still lacked security, and provided poor isolation between applications and the operating system. Thus, the problems with instability continued, even with the subsequent releases of **Windows 98** and **Windows Me**, where MS-DOS was still there running 16-bit assembly code in the heart of the Windows operating system.

11.1.3 2000s: NT-based Windows

By end of the 1980s, Microsoft realized that continuing to evolve an operating system with MS-DOS at its center was not the best way to go. PC hardware was continuing to increase in speed and capability and ultimately the PC market would collide with the desktop, workstation, and enterprise-server computing markets, where UNIX was the dominant operating system. Microsoft was also concerned that the Intel microprocessor family might not continue to be competitive, as it was already being challenged by RISC architectures. To address these issues, Microsoft recruited a group of engineers from DEC (Digital Equipment Corporation) led by Dave Cutler, one of the key designers of DEC's VMS operating system (among others). Cutler was chartered to develop a brand-new 32-bit operating system that was intended to implement **OS/2**, the operating system API that Microsoft was jointly developing with IBM at the time. The original design documents by Cutler's team called the system **NT OS/2**.

Cutler's system was called **NT** for New Technology (and also because the original target processor was the new Intel 860, code-named the N10). NT was designed to be portable across different processors and emphasized security and reliability, as well as compatibility with the MS-DOS-based versions of Windows. Cutler's background at DEC shows in various places, with there being more than a passing similarity between the design of NT and that of VMS and other operating systems designed by Cutler, shown in Fig. 11-2.

Year	DEC operating system	Characteristics
1973	RSX-11M	16-bit, multiuser, real-time, swapping
1978	VAX/VMS	32-bit, virtual memory
1987	VAXELAN	Real-time
1988	PRISM/Mica	Canceled in favor of MIPS/Ultrix

Figure 11-2. DEC operating systems developed by Dave Cutler.

Programmers familiar only with UNIX find the architecture of NT to be quite different. This is not just because of the influence of VMS, but also because of the differences in the computer systems that were common at the time of design. UNIX was first designed in the 1970s for single-processor, 16-bit, tiny-memory, swapping systems where the process was the unit of concurrency and composition, and fork/exec were inexpensive operations (since swapping systems frequently copy processes to disk anyway). NT was designed in the early 1990s, when multiprocessor, 32-bit, multimegabyte, virtual memory systems were common. In NT, threads are the units of concurrency, dynamic libraries are the units of composition, and fork/exec are implemented by a single operation to create a new process *and* run another program without first making a copy.

The first version of NT-based Windows (Windows NT 3.1) was released in 1993. It was called 3.1 to correspond with the then-current consumer Windows 3.1. The joint project with IBM had foundered, so though the OS/2 interfaces were still supported, the primary interfaces were 32-bit extensions of the Windows APIs, called **Win32**. Between the time NT was started and first shipped, Windows 3.0 had been released and had become extremely successful commercially. It too was able to run Win32 programs, but using the *Win32s* compatibility library.

Like the first version of MS-DOS-based Windows, NT-based Windows was not initially successful. NT required more memory, there were few 32-bit applications available, and incompatibilities with device drivers and applications caused many customers to stick with MS-DOS-based Windows which Microsoft was still improving, releasing Windows 95 in 1995. Windows 95 provided native 32-bit programming interfaces like NT, but better compatibility with existing 16-bit software and applications. Not surprisingly, NT's early success was in the server market, competing with VMS and NetWare.

NT did meet its portability goals, with additional releases in 1994 and 1995 adding support for (little-endian) MIPS and PowerPC architectures. The first major upgrade to NT came with **Windows NT 4.0** in 1996. This system had the power, security, and reliability of NT, but also sported the same user interface as the by-then very popular Windows 95.

Figure 11-3 shows the relationship of the Win32 API to Windows. Having a common API across both the MS-DOS-based and NT-based Windows was important to the success of NT.

This compatibility made it much easier for users to migrate from Windows 95 to NT, and the operating system became a strong player in the high-end desktop market as well as servers. However, customers were not as willing to adopt other processor architectures, and of the four architectures Windows NT 4.0 supported in 1996 (the DEC Alpha was added in that release), only the x86 (i.e., Pentium family) was still actively supported by the time of the next major release, **Windows 2000**.

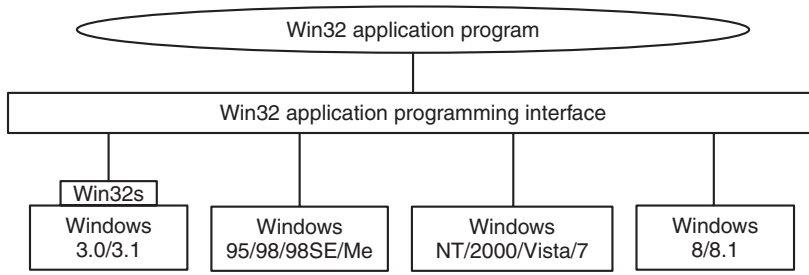

Figure 11-3. The Win32 API allows programs to run on almost all versions of Windows.

Windows 2000 represented a significant evolution for NT. The key technologies added were plug-and-play (for consumers who installed a new PCI card, eliminating the need to fiddle with jumpers), network directory services (for enterprise customers), improved power management (for notebook computers), and an improved GUI (for everyone).

The technical success of Windows 2000 led Microsoft to push toward the deprecation of Windows 98 by enhancing the application and device compatibility of the next NT release, **Windows XP**. Windows XP included a friendlier new look-and-feel to the graphical interface, bolstering Microsoft's strategy of hooking consumers and reaping the benefit as they pressured their employers to adopt systems with which they were already familiar. The strategy was overwhelmingly successful, with Windows XP being installed on hundreds of millions of PCs over its first few years, allowing Microsoft to achieve its goal of effectively ending the era of MS-DOS-based Windows.

Microsoft followed up Windows XP by embarking on an ambitious release to kindle renewed excitement among PC consumers. The result, **Windows Vista**, was completed in late 2006, more than five years after Windows XP shipped. Windows Vista boasted yet another redesign of the graphical interface, and new security features under the covers. Most of the changes were in customer-visible experiences and capabilities. The technologies under the covers of the system improved incrementally, with much clean-up of the code and many improvements in performance, scalability, and reliability. The server version of Vista (Windows Server 2008) was delivered about a year after the consumer version. It shares, with Vista, the same core system components, such as the kernel, drivers, and low-level libraries and programs.

The human story of the early development of NT is related in the book *Showstopper* (Zachary, 1994). The book tells a lot about the key people involved and the difficulties of undertaking such an ambitious software development project.

11.1.4 Windows Vista

The release of Windows Vista culminated Microsoft's most extensive operating system project to date. The initial plans were so ambitious that a couple of years into its development Vista had to be restarted with a smaller scope. Plans to rely heavily on Microsoft's type-safe, garbage-collected .NET language C# were shelved, as were some significant features such as the WinFS unified storage system for searching and organizing data from many different sources. The size of the full operating system is staggering. The original NT release of 3 million lines of C/C++ that had grown to 16 million in NT 4, 30 million in 2000, and 50 million in XP. It is over 70 million lines in Vista and more in Windows 7 and 8.

Much of the size is due to Microsoft's emphasis on adding many new features to its products in every release. In the main *system32* directory, there are 1600 **DLLs** (**Dynamic Link Libraries**) and 400 **EXEs** (**Executables**), and that does not include the other directories containing the myriad of applets included with the operating system that allow users to surf the Web, play music and video, send email, scan documents, organize photos, and even make movies. Because Microsoft wants customers to switch to new versions, it maintains compatibility by generally keeping all the features, APIs, *applets* (small applications), etc., from the previous version. Few things ever get deleted. The result is that Windows was growing dramatically release to release. Windows' distribution media had moved from floppy, to CD, and with Windows Vista, to DVD. Technology had been keeping up, however, and faster processors and larger memories made it possible for computers to get faster despite all this bloat.

Unfortunately for Microsoft, Windows Vista was released at a time when customers were becoming enthralled with inexpensive computers, such as low-end notebooks and **netbook** computers. These machines used slower processors to save cost and battery life, and in their earlier generations limited memory sizes. At

the same time, processor performance ceased to improve at the same rate it had previously, due to the difficulties in dissipating the heat created by ever-increasing clock speeds. Moore's Law continued to hold, but the additional transistors were going into new features and multiple processors rather than improvements in single-processor performance. All the bloat in Windows Vista meant that it performed poorly on these computers relative to Windows XP, and the release was never widely accepted.

The issues with Windows Vista were addressed in the subsequent release, **Windows 7**. Microsoft invested heavily in testing and performance automation, new telemetry technology, and extensively strengthened the teams charged with improving performance, reliability, and security. Though Windows 7 had relatively few functional changes compared to Windows Vista, it was better engineered and more efficient. Windows 7 quickly supplanted Vista and ultimately Windows XP to be the most popular version of Windows to date.

11.1.5 2010s: Modern Windows

By the time Windows 7 shipped, the computing industry once again began to change dramatically. The success of the Apple iPhone as a portable computing device, and the advent of the Apple iPad, had heralded a sea-change which led to the dominance of lower-cost Android tablets and phones, much as Microsoft had dominated the desktop in the first three decades of personal computing. Small, portable, yet powerful devices and ubiquitous fast networks were creating a world where mobile computing and network-based services were becoming the dominant paradigm. The old world of portable computers was replaced by machines with small screens that ran applications readily downloadable from the Web. These applications were not the traditional variety, like word processing, spreadsheets, and connecting to corporate servers. Instead, they provided access to services like Web search, social networking, Wikipedia, streaming music and video, shopping, and personal navigation. The business models for computing were also changing, with advertising opportunities becoming the largest economic force behind computing.

Microsoft began a process to redesign itself as a *devices and services* company in order to better compete with Google and Apple. It needed an operating system it could deploy across a wide spectrum of devices: phones, tablets, game consoles, laptops, desktops, servers, and the cloud. Windows thus underwent an even bigger evolution than with Windows Vista, resulting in **Windows 8**. However, this time Microsoft applied the lessons from Windows 7 to create a well-engineered, performant product with less bloat.

Windows 8 built on the modular **MinWin** approach Microsoft used in Windows 7 to produce a small operating system core that could be extended onto different devices. The goal was for each of the operating systems for specific devices to be built by extending this core with new user interfaces and features, yet provide as common an experience for users as possible. This approach was successfully

applied to Windows Phone 8, which shares most of the core binaries with desktop and server Windows. Support of phones and tablets by Windows required support for the popular ARM architecture, as well as new Intel processors targeting those devices. What makes Windows 8 part of the Modern Windows era are the fundamental changes in the programming models, as we will examine in the next section.

Windows 8 was not received to universal acclaim. In particular, the lack of the Start Button on the taskbar (and its associated menu) was viewed by many users as a huge mistake. Others objected to using a tablet-like interface on a desktop machine with a large monitor. Microsoft responded to this and other criticisms on May 14, 2013 by releasing an update called **Windows 8.1**. This version fixed these problems while at the same time introducing a host of new features, such as better cloud integration, as well as a number of new programs. Although we will stick to the more generic name of "Windows 8" in this chapter, in fact, everything in it is a description of how Windows 8.1 works.

11.2 PROGRAMMING WINDOWS

It is now time to start our technical study of Windows. Before getting into the details of the internal structure, however, we will take a look at the native NT API for system calls, the Win32 programming subsystem introduced as part of NT-based Windows, and the Modern WinRT programming environment introduced with Windows 8.

Figure 11-4 shows the layers of the Windows operating system. Beneath the applet and GUI layers of Windows are the programming interfaces that applications build on. As in most operating systems, these consist largely of code libraries (DLLs) to which programs dynamically link for access to operating system features. Windows also includes a number of programming interfaces which are implemented as services that run as separate processes. Applications communicate with user-mode services through **RPCs** (**Remote-Procedure-Calls**).

The core of the NT operating system is the **NTOS** kernel-mode program (*ntoskrnl.exe*), which provides the traditional system-call interfaces upon which the rest of the operating system is built. In Windows, only programmers at Microsoft write to the system-call layer. The published user-mode interfaces all belong to operating system personalities that are implemented using **subsystems** that run on top of the NTOS layers.

Originally NT supported three personalities: OS/2, POSIX and Win32. OS/2 was discarded in Windows XP. Support for POSIX was finally removed in Windows 8.1. Today all Windows applications are written using APIs that are built on top of the Win32 subsystem, such as the WinFX API in the .NET programming model. The WinFX API includes many of the features of Win32, and in fact many

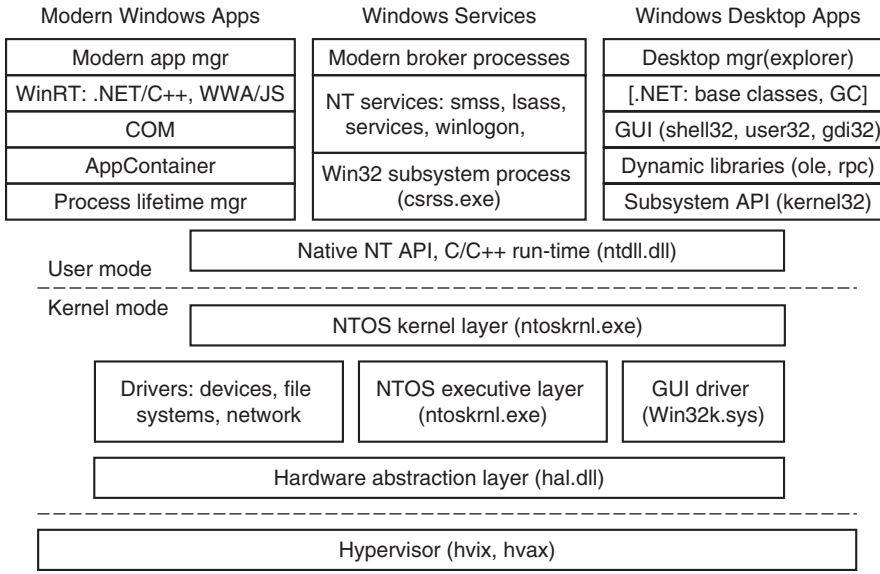

Figure 11-4. The programming layers in Modern Windows.

of the functions in the WinFX *Base Class Library* are simply wrappers around Win32 APIs. The advantages of WinFX have to do with the richness of the object types supported, the simplified consistent interfaces, and use of the .NET Common Language Run-time (CLR), including garbage collection (GC).

The Modern versions of Windows begin with Windows 8, which introduced the new **WinRT** set of APIs. Windows 8 deprecated the traditional Win32 desktop experience in favor of running a single application at a time on the full screen with an emphasis on touch over use of the mouse. Microsoft saw this as a necessary step as part of the transition to a single operating system that would work with phones, tablets, and game consoles, as well as traditional PCs and servers. The GUI changes necessary to support this new model require that applications be rewritten to a new API model, the **Modern Software Development Kit**, which includes the WinRT APIs. The WinRT APIs are carefully curated to produce a more consistent set of behaviors and interfaces. These APIs have versions available for C++ and .NET programs but also JavaScript for applications hosted in a browser-like environment *wwa.exe* (Windows Web Application).

In addition to WinRT APIs, many of the existing Win32 APIs were included in the **MSDK** (**Microsoft Development Kit**). The initially available WinRT APIs were not sufficient to write many programs. Some of the included Win32 APIs were chosen to limit the behavior of applications. For example, applications cannot create threads directly with the MSDK, but must rely on the Win32 thread pool to run concurrent activities within a process. This is because Modern Windows is

shifting programmers away from a threading model to a task model in order to disentangle resource management (priorities, processor affinities) from the programming model (specifying concurrent activities). Other omitted Win32 APIs include most of the Win32 virtual memory APIs. Programmers are expected to rely on the Win32 heap-management APIs rather than attempt to manage memory resources directly. APIs that were already deprecated in Win32 were also omitted from the MSDK, as were all ANSI APIs. The MSDK APIs are Unicode only.

The choice of the word *Modern* to describe a product such as Windows is surprising. Perhaps if a new generation Windows is here ten years from now, it will be referred to as *post-Modern* Windows.

Unlike traditional Win32 processes, the processes running modern applications have their lifetimes managed by the operating system. When a user switches away from an application, the system gives it a couple of seconds to save its state and then ceases to give it further processor resources until the user switches back to the application. If the system runs low on resources, the operating system may terminate the application's processes without the application ever running again. When the user switches back to the application at some time in the future, it will be restarted by the operating system. Applications that need to run tasks in the background must specifically arrange to do so using a new set of WinRT APIs. Background activity is carefully managed by the system to improve battery life and prevent interference with the foreground application the user is currently using. These changes were made to make Windows function better on mobile devices.

In the Win32 desktop world applications are deployed by running an installer that is part of the application. Modern applications have to be installed using Windows' AppStore program, which will deploy only applications that were uploaded into the Microsoft on-line store by the developer. Microsoft is following the same successful model introduced by Apple and adopted by Android. Microsoft will not accept applications into the store unless they pass verification which, among other checks, ensures that the application is using only APIs available in the MSDK.

When a modern application is running, it always executes in a *sandbox* called an **AppContainer**. Sandboxing process execution is a security technique for isolating less trusted code so that it cannot freely tamper with the system or user data. The Windows AppContainer treats each application as a distinct user, and uses Windows security facilities to keep the application from accessing arbitrary system resources. When an application does need access to a system resource, there are WinRT APIs that communicate to **broker processes** which do have access to more of the system, such as a user's files.

As shown in Fig. 11-5, NT subsystems are constructed out of four components: a subsystem process, a set of libraries, hooks in CreateProcess, and support in the kernel. A subsystem process is really just a service. The only special property is that it is started by the *smss.exe* (session manager) program—the initial user-mode program started by NT—in response to a request from **CreateProcess** in Win32 or the corresponding API in a different subsystem. Although Win32 is

the only remaining subsystem supported, Windows still maintains the subsystem model, including the *csrss.exe* Win32 subsystem process.

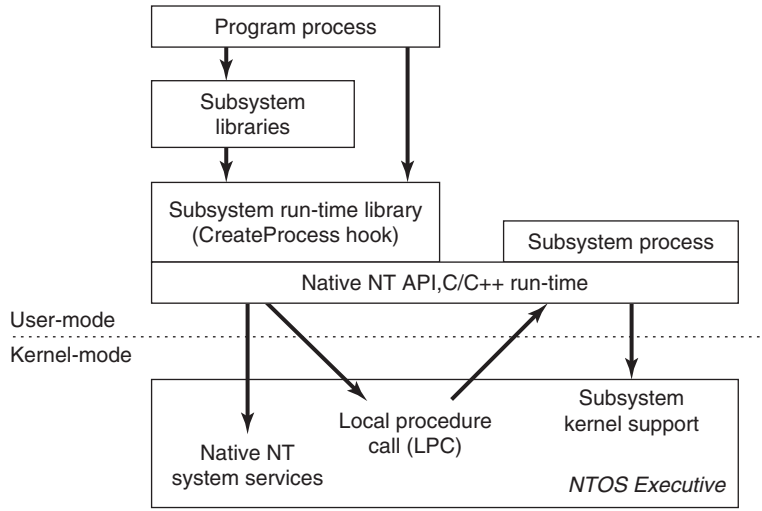

Figure 11-5. The components used to build NT subsystems.

The set of libraries both implements higher-level operating-system functions specific to the subsystem and contains the stub routines which communicate between processes using the subsystem (shown on the left) and the subsystem process itself (shown on the right). Calls to the subsystem process normally take place using the kernel-mode **LPC** (**Local Procedure Call**) facilities, which implement cross-process procedure calls.

The hook in Win32 CreateProcess detects which subsystem each program requires by looking at the binary image. It then asks *smss.exe* to start the subsystem process (if it is not already running). The subsystem process then takes over responsibility for loading the program.

The NT kernel was designed to have a lot of general-purpose facilities that can be used for writing operating-system-specific subsystems. But there is also special code that must be added to correctly implement each subsystem. As examples, the native NtCreateProcess system call implements process duplication in support of POSIX fork system call, and the kernel implements a particular kind of string table for Win32 (called *atoms*) which allows read-only strings to be efficiently shared across processes.

The subsystem processes are native NT programs which use the native system calls provided by the NT kernel and core services, such as *smss.exe* and *lsass.exe* (local security administration). The native system calls include cross-process facilities to manage virtual addresses, threads, handles, and exceptions in the processes created to run programs written to use a particular subsystem.

11.2.1 The Native NT Application Programming Interface

Like all other operating systems, Windows has a set of system calls it can perform. In Windows, these are implemented in the NTOS executive layer that runs in kernel mode. Microsoft has published very few of the details of these native system calls. They are used internally by lower-level programs that ship as part of the operating system (mainly services and the subsystems), as well as kernel-mode device drivers. The native NT system calls do not really change very much from release to release, but Microsoft chose not to make them public so that applications written for Windows would be based on Win32 and thus more likely to work with both the MS-DOS-based and NT-based Windows systems, since the Win32 API is common to both.

Most of the native NT system calls operate on kernel-mode objects of one kind or another, including files, processes, threads, pipes, semaphores, and so on. Figure 11-6 gives a list of some of the common categories of kernel-mode objects supported by the kernel in Windows. Later, when we discuss the object manager, we will provide further details on the specific object types.

Object category	Examples
Synchronization	Semaphores, mutexes, events, IPC ports, I/O completion queues
I/O	Files, devices, drivers, timers
Program	Jobs, processes, threads, sections, tokens
Win32 GUI	Desktops, application callbacks

Figure 11-6. Common categories of kernel-mode object types.

Sometimes use of the term *object* regarding the data structures manipulated by the operating system can be confusing because it is mistaken for *object-oriented*. Operating system objects do provide data hiding and abstraction, but they lack some of the most basic properties of object-oriented systems such as inheritance and polymorphism.

In the native NT API, calls are available to create new kernel-mode objects or access existing ones. Every call creating or opening an object returns a result called a **handle** to the caller. The handle can subsequently be used to perform operations on the object. Handles are specific to the process that created them. In general handles cannot be passed directly to another process and used to refer to the same object. However, under certain circumstances, it is possible to duplicate a handle into the handle table of other processes in a protected way, allowing processes to share access to objects—even if the objects are not accessible in the namespace. The process duplicating each handle must itself have handles for both the source and target process.

Every object has a **security descriptor** associated with it, telling in detail who may and may not perform what kinds of operations on the object based on the

access requested. When handles are duplicated between processes, new access restrictions can be added that are specific to the duplicated handle. Thus, a process can duplicate a read-write handle and turn it into a read-only version in the target process.

Not all system-created data structures are objects and not all objects are kernel-mode objects. The only ones that are true kernel-mode objects are those that need to be named, protected, or shared in some way. Usually, they represent some kind of programming abstraction implemented in the kernel. Every kernel-mode object has a system-defined type, has well-defined operations on it, and occupies storage in kernel memory. Although user-mode programs can perform the operations (by making system calls), they cannot get at the data directly.

Figure 11-7 shows a sampling of the native APIs, all of which use explicit handles to manipulate kernel-mode objects such as processes, threads, IPC ports, and **sections** (which are used to describe memory objects that can be mapped into address spaces). NtCreateProcess returns a handle to a newly created process object, representing an executing instance of the program represented by the SectionHandle. DebugPortHandle is used to communicate with a debugger when giving it control of the process after an exception (e.g., dividing by zero or accessing invalid memory). ExceptPortHandle is used to communicate with a subsystem process when errors occur and are not handled by an attached debugger.

NtCreateProcess(&ProcHandle, Access, SectionHandle, DebugPortHandle, ExceptPortHandle, ...)
NtCreateThread(&ThreadHandle, ProcHandle, Access, ThreadContext, CreateSuspended, ...)
NtAllocateVirtualMemory(ProcHandle, Addr, Size, Type, Protection, ...)
NtMapViewOfSection(SectHandle, ProcHandle, Addr, Size, Protection, ...)
NtReadVirtualMemory(ProcHandle, Addr, Size, ...)
NtWriteVirtualMemory(ProcHandle, Addr, Size, ...)
NtCreateFile(&FileHandle, FileNameDescriptor, Access, ...)
NtDuplicateObject(srcProcHandle, srcObjHandle, dstProcHandle, dstObjHandle, ...)

Figure 11-7. Examples of native NT API calls that use handles to manipulate objects across process boundaries.

NtCreateThread takes ProcHandle because it can create a thread in any process for which the calling process has a handle (with sufficient access rights). Similarly, NtAllocateVirtualMemory, NtMapViewOfSection, NtReadVirtualMemory, and NtWriteVirtualMemory allow one process not only to operate on its own address space, but also to allocate virtual addresses, map sections, and read or write virtual memory in other processes. NtCreateFile is the native API call for creating a new file or opening an existing one. NtDuplicateObject is the API call for duplicating handles from one process to another.

Kernel-mode objects are, of course, not unique to Windows. UNIX systems also support a variety of kernel-mode objects, such as files, network sockets, pipes,

devices, processes, and interprocess communication (IPC) facilities like shared memory, message ports, semaphores, and I/O devices. In UNIX there are a variety of ways of naming and accessing objects, such as file descriptors, process IDs, and integer IDs for SystemV IPC objects, and i-nodes for devices. The implementation of each class of UNIX objects is specific to the class. Files and sockets use different facilities than the SystemV IPC mechanisms or processes or devices.

Kernel objects in Windows use a uniform facility based on handles and names in the NT namespace to reference kernel objects, along with a unified implementation in a centralized **object manager**. Handles are per-process but, as described above, can be duplicated into another process. The object manager allows objects to be given names when they are created, and then opened by name to get handles for the objects.

The object manager uses **Unicode** (wide characters) to represent names in the **NT namespace**. Unlike UNIX, NT does not generally distinguish between upper- and lowercase (it is *case preserving* but *case insensitive*). The NT namespace is a hierarchical tree-structured collection of directories, symbolic links and objects.

The object manager also provides unified facilities for synchronization, security, and object lifetime management. Whether the general facilities provided by the object manager are made available to users of any particular object is up to the executive components, as they provide the native APIs that manipulate each object type.

It is not only applications that use objects managed by the object manager. The operating system itself can also create and use objects—and does so heavily. Most of these objects are created to allow one component of the system to store some information for a substantial period of time or to pass some data structure to another component, and yet benefit from the naming and lifetime support of the object manager. For example, when a device is discovered, one or more **device objects** are created to represent the device and to logically describe how the device is connected to the rest of the system. To control the device a device driver is loaded, and a **driver object** is created holding its properties and providing pointers to the functions it implements for processing the I/O requests. Within the operating system the driver is then referred to by using its object. The driver can also be accessed directly by name rather than indirectly through the devices it controls (e.g., to set parameters governing its operation from user mode).

Unlike UNIX, which places the root of its namespace in the file system, the root of the NT namespace is maintained in the kernel's virtual memory. This means that NT must recreate its top-level namespace every time the system boots. Using kernel virtual memory allows NT to store information in the namespace without first having to start the file system running. It also makes it much easier for NT to add new types of kernel-mode objects to the system because the formats of the file systems themselves do not have to be modified for each new object type.

A named object can be marked *permanent*, meaning that it continues to exist until explicitly deleted or the system reboots, even if no process currently has a

handle for the object. Such objects can even extend the NT namespace by providing *parse* routines that allow the objects to function somewhat like mount points in UNIX. File systems and the registry use this facility to mount volumes and hives onto the NT namespace. Accessing the device object for a volume gives access to the raw volume, but the device object also represents an implicit mount of the volume into the NT namespace. The individual files on a volume can be accessed by concatenating the volume-relative file name onto the end of the name of the device object for that volume.

Permanent names are also used to represent synchronization objects and shared memory, so that they can be shared by processes without being continually recreated as processes stop and start. Device objects and often driver objects are given permanent names, giving them some of the persistence properties of the special i-nodes kept in the */dev* directory of UNIX.

We will describe many more of the features in the native NT API in the next section, where we discuss the Win32 APIs that provide wrappers around the NT system calls.

11.2.2 The Win32 Application Programming Interface

The Win32 function calls are collectively called the **Win32 API**. These interfaces are publicly disclosed and fully documented. They are implemented as library procedures that either wrap the native NT system calls used to get the work done or, in some cases, do the work right in user mode. Though the native NT APIs are not published, most of the functionality they provide is accessible through the Win32 API. The existing Win32 API calls rarely change with new releases of Windows, though many new functions are added to the API.

Figure 11-8 shows various low-level Win32 API calls and the native NT API calls that they wrap. What is interesting about the figure is how uninteresting the mapping is. Most low-level Win32 functions have native NT equivalents, which is not surprising as Win32 was designed with NT in mind. In many cases the Win32 layer must manipulate the Win32 parameters to map them onto NT, for example, canonicalizing path names and mapping onto the appropriate NT path names, including special MS-DOS device names (like *LPT:*). The Win32 APIs for creating processes and threads also must notify the Win32 subsystem process, *csrss.exe*, that there are new processes and threads for it to supervise, as we will describe in Sec. 11.4.

Some Win32 calls take path names, whereas the equivalent NT calls use handles. So the wrapper routines have to open the files, call NT, and then close the handle at the end. The wrappers also translate the Win32 APIs from ANSI to Unicode. The Win32 functions shown in Fig. 11-8 that use strings as parameters are actually two APIs, for example, **CreateProcessW** and **CreateProcessA**. The strings passed to the latter API must be translated to Unicode before calling the underlying NT API, since NT works only with Unicode.

Win32 call	Native NT API call
CreateProcess	NtCreateProcess
CreateThread	NtCreateThread
SuspendThread	NtSuspendThread
CreateSemaphore	NtCreateSemaphore
ReadFile	NtReadFile
DeleteFile	NtSetInformationFile
CreateFileMapping	NtCreateSection
VirtualAlloc	NtAllocateVirtualMemory
MapViewOfFile	NtMapViewOfSection
DuplicateHandle	NtDuplicateObject
CloseHandle	NtClose

Figure 11-8. Examples of Win32 API calls and the native NT API calls that they wrap.

Since few changes are made to the existing Win32 interfaces in each release of Windows, in theory the binary programs that ran correctly on any previous release will continue to run correctly on a new release. In practice, there are often many compatibility problems with new releases. Windows is so complex that a few seemingly inconsequential changes can cause application failures. And applications themselves are often to blame, since they frequently make explicit checks for specific operating system versions or fall victim to their own latent bugs that are exposed when they run on a new release. Nevertheless, Microsoft makes an effort in every release to test a wide variety of applications to find incompatibilities and either correct them or provide application-specific workarounds.

Windows supports two special execution environments both called **WOW** (**Windows-on-Windows**). WOW32 is used on 32-bit x86 systems to run 16-bit Windows 3.x applications by mapping the system calls and parameters between the 16-bit and 32-bit worlds. Similarly, WOW64 allows 32-bit Windows applications to run on x64 systems.

The Windows API philosophy is very different from the UNIX philosophy. In the latter, the operating system functions are simple, with few parameters and few places where there are multiple ways to perform the same operation. Win32 provides very comprehensive interfaces with many parameters, often with three or four ways of doing the same thing, and mixing together low-level and high-level functions, like CreateFile and CopyFile.

This means Win32 provides a very rich set of interfaces, but it also introduces much complexity due to the poor layering of a system that intermixes both high-level and low-level functions in the same API. For our study of operating systems, only the low-level functions of the Win32 API that wrap the native NT API are relevant, so those are what we will focus on.

Win32 has calls for creating and managing both processes and threads. There are also many calls that relate to interprocess communication, such as creating, destroying, and using mutexes, semaphores, events, communication ports, and other IPC objects.

Although much of the memory-management system is invisible to programmers, one important feature is visible: namely the ability of a process to map a file onto a region of its virtual memory. This allows threads running in a process the ability to read and write parts of the file using pointers without having to explicitly perform read and write operations to transfer data between the disk and memory. With memory-mapped files the memory-management system itself performs the I/Os as needed (demand paging).

Windows implements memory-mapped files using three completely different facilities. First it provides interfaces which allow processes to manage their own virtual address space, including reserving ranges of addresses for later use. Second, Win32 supports an abstraction called a **file mapping**, which is used to represent addressable objects like files (a file mapping is called a *section* in the NT layer). Most often, file mappings are created to refer to files using a file handle, but they can also be created to refer to private pages allocated from the system pagefile.

The third facility maps *views* of file mappings into a process' address space. Win32 allows only a view to be created for the current process, but the underlying NT facility is more general, allowing views to be created for any process for which you have a handle with the appropriate permissions. Separating the creation of a file mapping from the operation of mapping the file into the address space is a different approach than used in the mmap function in UNIX.

In Windows, the file mappings are kernel-mode objects represented by a handle. Like most handles, file mappings can be duplicated into other processes. Each of these processes can map the file mapping into its own address space as it sees fit. This is useful for sharing private memory between processes without having to create files for sharing. At the NT layer, file mappings (sections) can also be made persistent in the NT namespace and accessed by name.

An important area for many programs is file I/O. In the basic Win32 view, a file is just a linear sequence of bytes. Win32 provides over 60 calls for creating and destroying files and directories, opening and closing files, reading and writing them, requesting and setting file attributes, locking ranges of bytes, and many more fundamental operations on both the organization of the file system and access to individual files.

There are also various advanced facilities for managing data in files. In addition to the primary data stream, files stored on the NTFS file system can have additional data streams. Files (and even entire volumes) can be encrypted. Files can be compressed, and/or represented as a sparse stream of bytes where missing regions of data in the middle occupy no storage on disk. File-system volumes can be organized out of multiple separate disk partitions using different levels of RAID

storage. Modifications to files or directory subtrees can be detected through a notification mechanism, or by reading the **journal** that NTFS maintains for each volume.

Each file-system volume is implicitly mounted in the NT namespace, according to the name given to the volume, so a file *\foo\bar* might be named, for example, *\Device\HarddiskVolume\foo\bar*. Internal to each NTFS volume, mount points (called *reparse points* in Windows) and symbolic links are supported to help organize the individual volumes.

The low-level I/O model in Windows is fundamentally asynchronous. Once an I/O operation is begun, the system call can return and allow the thread which initiated the I/O to continue in parallel with the I/O operation. Windows supports cancellation, as well as a number of different mechanisms for threads to synchronize with I/O operations when they complete. Windows also allows programs to specify that I/O should be synchronous when a file is opened, and many library functions, such as the C library and many Win32 calls, specify synchronous I/O for compatibility or to simplify the programming model. In these cases the executive will explicitly synchronize with I/O completion before returning to user mode.

Another area for which Win32 provides calls is security. Every thread is associated with a kernel-mode object, called a **token**, which provides information about the identity and privileges associated with the thread. Every object can have an **ACL** (**Access Control List**) telling in great detail precisely which users may access it and which operations they may perform on it. This approach provides for fine-grained security in which specific users can be allowed or denied specific access to every object. The security model is extensible, allowing applications to add new security rules, such as limiting the hours access is permitted.

The Win32 namespace is different than the native NT namespace described in the previous section. Only parts of the NT namespace are visible to Win32 APIs (though the entire NT namespace can be accessed through a Win32 hack that uses special prefix strings, like "*\\.*"). In Win32, files are accessed relative to *drive letters*. The NT directory *\DosDevices* contains a set of symbolic links from drive letters to the actual device objects. For example, *\DosDevices\C:* might be a link to *\Device\HarddiskVolume1*. This directory also contains links for other Win32 devices, such as *COM1:*, *LPT:*, and *NUL:* (for the serial and printer ports and the all-important null device). *\DosDevices* is really a symbolic link to *\??* which was chosen for efficiency. Another NT directory, *\BaseNamedObjects*, is used to store miscellaneous named kernel-mode objects accessible through the Win32 API. These include synchronization objects like semaphores, shared memory, timers, communication ports, and device names.

In addition to low-level system interfaces we have described, the Win32 API also supports many functions for GUI operations, including all the calls for managing the graphical interface of the system. There are calls for creating, destroying, managing, and using windows, menus, tool bars, status bars, scroll bars, dialog boxes, icons, and many more items that appear on the screen. There are calls for

drawing geometric figures, filling them in, managing the color palettes they use, dealing with fonts, and placing icons on the screen. Finally, there are calls for dealing with the keyboard, mouse and other human-input devices as well as audio, printing, and other output devices.

The GUI operations work directly with the *win32k.sys* driver using special interfaces to access these functions in kernel mode from user-mode libraries. Since these calls do not involve the core system calls in the NTOS executive, we will not say more about them.

11.2.3 The Windows Registry

The root of the NT namespace is maintained in the kernel. Storage, such as file-system volumes, is attached to the NT namespace. Since the NT namespace is constructed afresh every time the system boots, how does the system know about any specific details of the system configuration? The answer is that Windows attaches a special kind of file system (optimized for small files) to the NT namespace. This file system is called the **registry**. The registry is organized into separate volumes called **hives**. Each hive is kept in a separate file (in the directory *C:\Windows\system32\config* of the boot volume). When a Windows system boots, one particular hive named *SYSTEM* is loaded into memory by the same boot program that loads the kernel and other boot files, such as boot drivers, from the boot volume.

Windows keeps a great deal of crucial information in the SYSTEM hive, including information about what drivers to use with what devices, what software to run initially, and many parameters governing the operation of the system. This information is used even by the boot program itself to determine which drivers are boot drivers, being needed immediately upon boot. Such drivers include those that understand the file system and disk drivers for the volume containing the operating system itself.

Other configuration hives are used after the system boots to describe information about the software installed on the system, particular users, and the classes of user-mode **COM** (**Component Object-Model**) objects that are installed on the system. Login information for local users is kept in the **SAM** (**Security Access Manager**) hive. Information for network users is maintained by the *lsass* service in the security hive and coordinated with the network directory servers so that users can have a common account name and password across an entire network. A list of the hives used in Windows is shown in Fig. 11-9.

Prior to the introduction of the registry, configuration information in Windows was kept in hundreds of *.ini* (initialization) files spread across the disk. The registry gathers these files into a central store, which is available early in the process of booting the system. This is important for implementing Windows plug-and-play functionality. Unfortunately, the registry has become seriously disorganized over time as Windows has evolved. There are poorly defined conventions about how the

Hive file	Mounted name	Use
SYSTEM	HKLM\SYSTEM	OS configuration information, used by kernel
HARDWARE	HKLM\HARDWARE	In-memory hive recording hardware detected
BCD	HKLM\BCD*	Boot Configuration Database
SAM	HKLM\SAM	Local user account information
SECURITY	HKLM\SECURITY	lsass' account and other security information
DEFAULT	HKEY_USERS\.DEFAULT	Default hive for new users
NTUSER.DAT	HKEY_USERS\<user id>	User-specific hive, kept in home directory
SOFTWARE	HKLM\SOFTWARE	Application classes registered by COM
COMPONENTS	HKLM\COMPONENTS	Manifests and dependencies for sys. components

Figure 11-9. The registry hives in Windows. HKLM is a shorthand for *HKEY_LOCAL_MACHINE*.

configuration information should be arranged, and many applications take an ad hoc approach. Most users, applications, and all drivers run with full privileges and frequently modify system parameters in the registry directly—sometimes interfering with each other and destabilizing the system.

The registry is a strange cross between a file system and a database, and yet really unlike either. Entire books have been written describing the registry (Born, 1998; Hipson, 2002; and Ivens, 1998), and many companies have sprung up to sell special software just to manage the complexity of the registry.

To explore the registry Windows has a GUI program called **regedit** that allows you to open and explore the directories (called *keys*) and data items (called *values*). Microsoft's **PowerShell** scripting language can also be useful for walking through the keys and values of the registry as if they were directories and files. A more interesting tool to use is *procmon*, which is available from Microsoft's tools' Website: *www.microsoft.com/technet/sysinternals*.

Procmon watches all the registry accesses that take place in the system and is very illuminating. Some programs will access the same key over and over tens of thousands of times.

As the name implies, *regedit* allows users to edit the registry—but be very careful if you ever do. It is very easy to render your system unable to boot, or damage the installation of applications so that you cannot fix them without a lot of wizardry. Microsoft has promised to clean up the registry in future releases, but for now it is a huge mess—far more complicated than the configuration information maintained in UNIX. The complexity and fragility of the registry led designers of new operating systems—in particular—iOS and Android—to avoid anything like it.

The registry is accessible to the Win32 programmer. There are calls to create and delete keys, look up values within keys, and more. Some of the more useful ones are listed in Fig. 11-10.

Win32 API function	Description
RegCreateKeyEx	Create a new registry key
RegDeleteKey	Delete a registry key
RegOpenKeyEx	Open a key to get a handle to it
RegEnumKeyEx	Enumerate the subkeys subordinate to the key of the handle
RegQueryValueEx	Look up the data for a value within a key

Figure 11-10. Some of the Win32 API calls for using the registry

When the system is turned off, most of the registry information is stored on the disk in the hives. Because their integrity is so critical to correct system functioning, backups are made automatically and metadata writes are flushed to disk to prevent corruption in the event of a system crash. Loss of the registry requires reinstalling *all* software on the system.

11.3 SYSTEM STRUCTURE

In the previous sections we examined Windows as seen by the programmer writing code for user mode. Now we are going to look under the hood to see how the system is organized internally, what the various components do, and how they interact with each other and with user programs. This is the part of the system seen by the programmer implementing low-level user-mode code, like subsystems and native services, as well as the view of the system provided to device-driver writers.

Although there are many books on how to use Windows, there are many fewer on how it works inside. One of the best places to look for additional information on this topic is *Microsoft Windows Internals*, 6th ed., Parts 1 and 2 (Russinovich and Solomon, 2012).

11.3.1 Operating System Structure

As described earlier, the Windows operating system consists of many layers, as depicted in Fig. 11-4. In the following sections we will dig into the lowest levels of the operating system: those that run in kernel mode. The central layer is the NTOS kernel itself, which is loaded from *ntoskrnl.exe* when Windows boots. NTOS itself consists of two layers, the **executive**, which containing most of the services, and a smaller layer which is (also) called the **kernel** and implements the underlying thread scheduling and synchronization abstractions (a kernel within the kernel?), as well as implementing trap handlers, interrupts, and other aspects of how the CPU is managed.

The division of NTOS into kernel and executive is a reflection of NT's VAX/VMS roots. The VMS operating system, which was also designed by Cutler, had four hardware-enforced layers: user, supervisor, executive, and kernel corresponding to the four protection modes provided by the VAX processor architecture. The Intel CPUs also support four rings of protection, but some of the early target processors for NT did not, so the kernel and executive layers represent a software-enforced abstraction, and the functions that VMS provides in supervisor mode, such as printer spooling, are provided by NT as user-mode services.

The kernel-mode layers of NT are shown in Fig. 11-11. The kernel layer of NTOS is shown above the executive layer because it implements the trap and interrupt mechanisms used to transition from user mode to kernel mode.

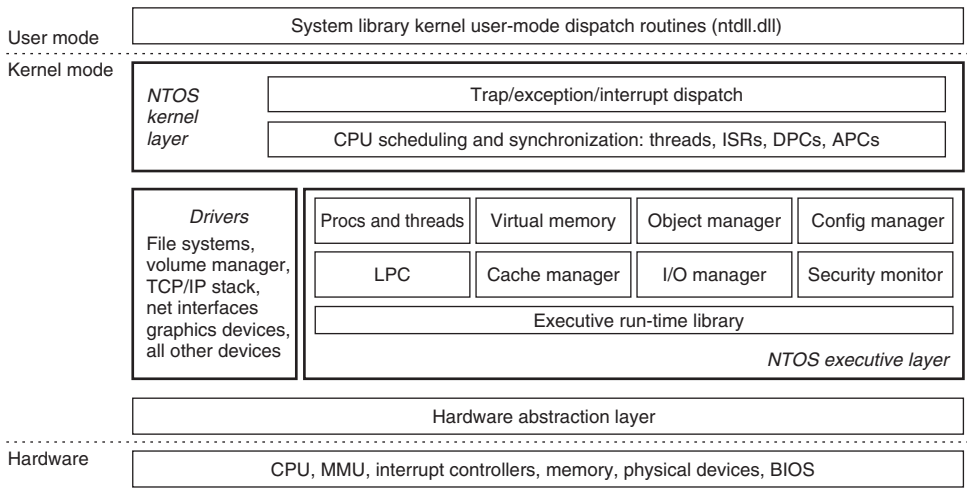

Figure 11-11. Windows kernel-mode organization.

The uppermost layer in Fig. 11-11 is the system library (*ntdll.dll*), which actually runs in user mode. The system library includes a number of support functions for the compiler run-time and low-level libraries, similar to what is in *libc* in UNIX. *ntdll.dll* also contains special code entry points used by the kernel to initialize threads and dispatch exceptions and user-mode **APCs** (**Asynchronous Procedure Calls**). Because the system library is so integral to the operation of the kernel, every user-mode process created by NTOS has *ntdll* mapped at the same fixed address. When NTOS is initializing the system it creates a section object to use when mapping *ntdll*, and it also records addresses of the *ntdll* entry points used by the kernel.

Below the NTOS kernel and executive layers is a layer of software called the **HAL** (**Hardware Abstraction Layer**) which abstracts low-level hardware details like access to device registers and DMA operations, and the way the parentboard

firmware represents configuration information and deals with differences in the CPU support chips, such as various interrupt controllers.

The lowest software layer is the **hypervisor**, which Windows calls **Hyper-V**. The hypervisor is an optional feature (not shown in Fig. 11-11). It is available in many versions of Windows—including the professional desktop client. The hypervisor intercepts many of the privileged operations performed by the kernel and emulates them in a way that allows multiple operating systems to run at the same time. Each operating system runs in its own virtual machine, which Windows calls a **partition**. The hypervisor uses features in the hardware architecture to protect physical memory and provide isolation between partitions. An operating system running on top of the hypervisor executes threads and handles interrupts on abstractions of the physical processors called **virtual processors**. The hypervisor schedules the virtual processors on the physical processors.

The main (root) operating system runs in the root partition. It provides many services to the other (guest) partitions. Some of the most important services provide integration of the guests with the shared devices such as networking and the GUI. While the root operating system must be Windows when running Hyper-V, other operating systems, such as Linux, can be run in the guest partitions. A guest operating system may perform very poorly unless it has been modified (i.e., paravirtualized) to work with the hypervisor.

For example, if a guest operating system kernel is using a spinlock to synchronize between two virtual processors and the hypervisor reschedules the virtual processor holding the spinlock, the lock hold time may increase by orders of magnitude, leaving other virtual processors running in the partition spinning for very long periods of time. To solve this problem a guest operating system is *enlightened* to spin only a short time before calling into the hypervisor to yield its physical processor to run another virtual processor.

The other major components of kernel mode are the device drivers. Windows uses device drivers for any kernel-mode facilities which are not part of NTOS or the HAL. This includes file systems, network protocol stacks, and kernel extensions like antivirus and **DRM** (**Digital Rights Management**) software, as well as drivers for managing physical devices, interfacing to hardware buses, and so on.

The I/O and virtual memory components cooperate to load (and unload) device drivers into kernel memory and link them to the NTOS and HAL layers. The I/O manager provides interfaces which allow devices to be discovered, organized, and operated—including arranging to load the appropriate device driver. Much of the configuration information for managing devices and drivers is maintained in the SYSTEM hive of the registry. The plug-and-play subcomponent of the I/O manager maintains information about the hardware detected within the HARDWARE hive, which is a volatile hive maintained in memory rather than on disk, as it is completely recreated every time the system boots.

We will now examine the various components of the operating system in a bit more detail.

The Hardware Abstraction Layer

One goal of Windows is to make the system portable across hardware platforms. Ideally, to bring up an operating system on a new type of computer system it should be possible to just recompile the operating system on the new platform. Unfortunately, it is not this simple. While many of the components in some layers of the operating system can be largely portable (because they mostly deal with internal data structures and abstractions that support the programming model), other layers must deal with device registers, interrupts, DMA, and other hardware features that differ significantly from machine to machine.

Most of the source code for the NTOS kernel is written in C rather than assembly language (only 2% is assembly on x86, and less than 1% on x64). However, all this C code cannot just be scooped up from an x86 system, plopped down on, say, an ARM system, recompiled, and rebooted owing to the many hardware differences between processor architectures that have nothing to do with the different instruction sets and which cannot be hidden by the compiler. Languages like C make it difficult to abstract away some hardware data structures and parameters, such as the format of page-table entries and the physical memory page sizes and word length, without severe performance penalties. All of these, as well as a slew of hardware-specific optimizations, would have to be manually ported even though they are not written in assembly code.

Hardware details about how memory is organized on large servers, or what hardware synchronization primitives are available, can also have a big impact on higher levels of the system. For example, NT's virtual memory manager and the kernel layer are aware of hardware details related to cache and memory locality. Throughout the system NT uses compare&swap synchronization primitives, and it would be difficult to port to a system that does not have them. Finally, there are many dependencies in the system on the ordering of bytes within words. On all the systems NT has ever been ported to, the hardware was set to little-endian mode.

Besides these larger issues of portability, there are also minor ones even between different parentboards from different manufacturers. Differences in CPU versions affect how synchronization primitives like spin-locks are implemented. There are several families of support chips that create differences in how hardware interrupts are prioritized, how I/O device registers are accessed, management of DMA transfers, control of the timers and real-time clock, multiprocessor synchronization, working with firmware facilities such as **ACPI** (**Advanced Configuration and Power Interface**), and so on. Microsoft made a serious attempt to hide these types of machine dependencies in a thin layer at the bottom called the HAL, as mentioned earlier. The job of the HAL is to present the rest of the operating system with abstract hardware that hides the specific details of processor version, support chipset, and other configuration variations. These HAL abstractions are presented in the form of machine-independent services (procedure calls and macros) that NTOS and the drivers can use.

By using the HAL services and not addressing the hardware directly, drivers and the kernel require fewer changes when being ported to new processors—and in most cases can run unmodified on systems with the same processor architecture, despite differences in versions and support chips.

The HAL does not provide abstractions or services for specific I/O devices such as keyboards, mice, and disks or for the memory management unit. These facilities are spread throughout the kernel-mode components, and without the HAL the amount of code that would have to be modified when porting would be substantial, even when the actual hardware differences were small. Porting the HAL itself is straightforward because all the machine-dependent code is concentrated in one place and the goals of the port are well defined: implement all of the HAL services. For many releases Microsoft supported a *HAL Development Kit* allowing system manufacturers to build their own HAL, which would allow other kernel components to work on new systems without modification, provided that the hardware changes were not too great.

As an example of what the hardware abstraction layer does, consider the issue of memory-mapped I/O vs. I/O ports. Some machines have one and some have the other. How should a driver be programmed: to use memory-mapped I/O or not? Rather than forcing a choice, which would make the driver not portable to a machine that did it the other way, the hardware abstraction layer offers three procedures for driver writers to use for reading the device registers and another three for writing them:

```
uc = READ_PORT_UCHAR(port);          WRITE_PORT_UCHAR(port, uc);
us = READ_PORT_USHORT(port);         WRITE_PORT_USHORT(port, us);
ul = READ_PORT_ULONG(port);          WRITE_PORT_LONG(port, ul);
```

These procedures read and write unsigned 8-, 16-, and 32-bit integers, respectively, to the specified port. It is up to the hardware abstraction layer to decide whether memory-mapped I/O is needed here. In this way, a driver can be moved without modification between machines that differ in the way the device registers are implemented.

Drivers frequently need to access specific I/O devices for various purposes. At the hardware level, a device has one or more addresses on a certain bus. Since modern computers often have multiple buses (PCI, PCIe, USB, IEEE 1394, etc.), it can happen that more than one device may have the same address on different buses, so some way is needed to distinguish them. The HAL provides a service for identifying devices by mapping bus-relative device addresses onto systemwide logical addresses. In this way, drivers do not have to keep track of which device is connected to which bus. This mechanism also shields higher layers from properties of alternative bus structures and addressing conventions.

Interrupts have a similar problem—they are also bus dependent. Here, too, the HAL provides services to name interrupts in a systemwide way and also provides ways to allow drivers to attach interrupt service routines to interrupts in a portable

way, without having to know anything about which interrupt vector is for which bus. Interrupt request level management is also handled in the HAL.

Another HAL service is setting up and managing DMA transfers in a device-independent way. Both the systemwide DMA engine and DMA engines on specific I/O cards can be handled. Devices are referred to by their logical addresses. The HAL implements software scatter/gather (writing or reading from noncontiguous blocks of physical memory).

The HAL also manages clocks and timers in a portable way. Time is kept track of in units of 100 nanoseconds starting at midnight on 1 January 1601, which is the first date in the previous quadricentury, which simplifies leap-year computations. (Quick Quiz: Was 1800 a leap year? Quick Answer: No.) The time services decouple the drivers from the actual frequencies at which the clocks run.

Kernel components sometimes need to synchronize at a very low level, especially to prevent race conditions in multiprocessor systems. The HAL provides primitives to manage this synchronization, such as spin locks, in which one CPU simply waits for a resource held by another CPU to be released, particularly in situations where the resource is typically held only for a few machine instructions.

Finally, after the system has been booted, the HAL talks to the computer's firmware (BIOS) and inspects the system configuration to find out which buses and I/O devices the system contains and how they have been configured. This information is then put into the registry. A summary of some of the things the HAL does is given in Fig. 11-12.

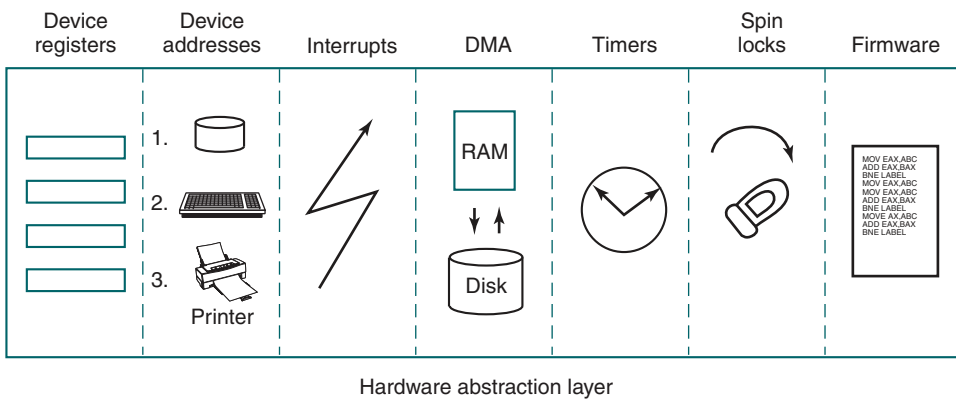

Figure 11-12. Some of the hardware functions the HAL manages.

The Kernel Layer

Above the hardware abstraction layer is NTOS, consisting of two layers: the kernel and the executive. "Kernel" is a confusing term in Windows. It can refer to all the code that runs in the processor's kernel mode. It can also refer to the

ntoskrnl.exe file which contains NTOS, the core of the Windows operating system. Or it can refer to the kernel layer within NTOS, which is how we use it in this section. It is even used to name the user-mode Win32 library that provides the wrappers for the native system calls: *kernel32.dll*.

In the Windows operating system the kernel layer, illustrated above the executive layer in Fig. 11-11, provides a set of abstractions for managing the CPU. The most central abstraction is threads, but the kernel also implements exception handling, traps, and several kinds of interrupts. Creating and destroying the data structures which support threading is implemented in the executive layer. The kernel layer is responsible for scheduling and synchronization of threads. Having support for threads in a separate layer allows the executive layer to be implemented using the same preemptive multithreading model used to write concurrent code in user mode, though the synchronization primitives in the executive are much more specialized.

The kernel's thread scheduler is responsible for determining which thread is executing on each CPU in the system. Each thread executes until a timer interrupt signals that it is time to switch to another thread (quantum expired), or until the thread needs to wait for something to happen, such as an I/O to complete or for a lock to be released, or a higher-priority thread becomes runnable and needs the CPU. When switching from one thread to another, the scheduler runs on the CPU and ensures that the registers and other hardware state have been saved. The scheduler then selects another thread to run on the CPU and restores the state that was previously saved from the last time that thread ran.

If the next thread to be run is in a different address space (i.e., process) than the thread being switched from, the scheduler must also change address spaces. The details of the scheduling algorithm itself will be discussed later in this chapter when we come to processes and threads.

In addition to providing a higher-level abstraction of the hardware and handling thread switches, the kernel layer also has another key function: providing low-level support for two classes of synchronization mechanisms: control objects and dispatcher objects. **Control objects** are the data structures that the kernel layer provides as abstractions to the executive layer for managing the CPU. They are allocated by the executive but they are manipulated with routines provided by the kernel layer. **Dispatcher objects** are the class of ordinary executive objects that use a common data structure for synchronization.

Deferred Procedure Calls

Control objects include primitive objects for threads, interrupts, timers, synchronization, profiling, and two special objects for implementing DPCs and APCs. **DPC** (**Deferred Procedure Call**) objects are used to reduce the time taken to execute **ISRs** (**Interrupt Service Routines**) in response to an interrupt from a particular device. Limiting time spent in ISRs reduces the chance of losing an interrupt.

The system hardware assigns a hardware priority level to interrupts. The CPU also associates a priority level with the work it is performing. The CPU responds only to interrupts at a higher-priority level than it is currently using. Normal priority levels, including the priority level of all user-mode work, is 0. Device interrupts occur at priority 3 or higher, and the ISR for a device interrupt normally executes at the same priority level as the interrupt in order to keep other less important interrupts from occurring while it is processing a more important one.

If an ISR executes too long, the servicing of lower-priority interrupts will be delayed, perhaps causing data to be lost or slowing the I/O throughput of the system. Multiple ISRs can be in progress at any one time, with each successive ISR being due to interrupts at higher and higher-priority levels.

To reduce the time spent processing ISRs, only the critical operations are performed, such as capturing the result of an I/O operation and reinitializing the device. Further processing of the interrupt is deferred until the CPU priority level is lowered and no longer blocking the servicing of other interrupts. The DPC object is used to represent the further work to be done and the ISR calls the kernel layer to queue the DPC to the list of DPCs for a particular processor. If the DPC is the first on the list, the kernel registers a special request with the hardware to interrupt the CPU at priority 2 (which NT calls DISPATCH level). When the last of any executing ISRs completes, the interrupt level of the processor will drop back below 2, and that will unblock the interrupt for DPC processing. The ISR for the DPC interrupt will process each of the DPC objects that the kernel had queued.

The technique of using software interrupts to defer interrupt processing is a well-established method of reducing ISR latency. UNIX and other systems started using deferred processing in the 1970s to deal with the slow hardware and limited buffering of serial connections to terminals. The ISR would deal with fetching characters from the hardware and queuing them. After all higher-level interrupt processing was completed, a software interrupt would run a low-priority ISR to do character processing, such as implementing backspace by sending control characters to the terminal to erase the last character displayed and move the cursor backward.

A similar example in Windows today is the keyboard device. After a key is struck, the keyboard ISR reads the key code from a register and then reenables the keyboard interrupt but does not do further processing of the key immediately. Instead, it uses a DPC to queue the processing of the key code until all outstanding device interrupts have been processed.

Because DPCs run at level 2 they do not keep device ISRs from executing, but they do prevent any threads from running until all the queued DPCs complete and the CPU priority level is lowered below 2. Device drivers and the system itself must take care not to run either ISRs or DPCs for too long. Because threads are not allowed to execute, ISRs and DPCs can make the system appear sluggish and produce glitches when playing music by stalling the threads writing the music buffer to the sound device. Another common use of DPCs is running routines in

response to a timer interrupt. To avoid blocking threads, timer events which need to run for an extended time should queue requests to the pool of worker threads the kernel maintains for background activities.

Asynchronous Procedure Calls

The other special kernel control object is the **APC** (**Asynchronous Procedure Call**) object. APCs are like DPCs in that they defer processing of a system routine, but unlike DPCs, which operate in the context of particular CPUs, APCs execute in the context of a specific thread. When processing a key press, it does not matter which context the DPC runs in because a DPC is simply another part of interrupt processing, and interrupts only need to manage the physical device and perform thread-independent operations such as recording the data in a buffer in kernel space.

The DPC routine runs in the context of whatever thread happened to be running when the original interrupt occurred. It calls into the I/O system to report that the I/O operation has been completed, and the I/O system queues an APC to run in the context of the thread making the original I/O request, where it can access the user-mode address space of the thread that will process the input.

At the next convenient time the kernel layer delivers the APC to the thread and schedules the thread to run. An APC is designed to look like an unexpected procedure call, somewhat similar to signal handlers in UNIX. The kernel-mode APC for completing I/O executes in the context of the thread that initiated the I/O, but in kernel mode. This gives the APC access to both the kernel-mode buffer as well as all of the user-mode address space belonging to the process containing the thread. *When* an APC is delivered depends on what the thread is already doing, and even what type of system. In a multiprocessor system the thread receiving the APC may begin executing even before the DPC finishes running.

User-mode APCs can also be used to deliver notification of I/O completion in user mode to the thread that initiated the I/O. User-mode APCs invoke a user-mode procedure designated by the application, but only when the target thread has blocked in the kernel and is marked as willing to accept APCs. The kernel interrupts the thread from waiting and returns to user mode, but with the user-mode stack and registers modified to run the APC dispatch routine in the *ntdll.dll* system library. The APC dispatch routine invokes the user-mode routine that the application has associated with the I/O operation. Besides specifying user-mode APCs as a means of executing code when I/Os complete, the Win32 API QueueUserAPC allows APCs to be used for arbitrary purposes.

The executive layer also uses APCs for operations other than I/O completion. Because the APC mechanism is carefully designed to deliver APCs only when it is safe to do so, it can be used to safely terminate threads. If it is not a good time to terminate the thread, the thread will have declared that it was entering a critical region and defer deliveries of APCs until it leaves. Kernel threads mark themselves

as entering critical regions to defer APCs when acquiring locks or other resources, so that they cannot be terminated while still holding the resource.

Dispatcher Objects

Another kind of synchronization object is the **dispatcher object**. This is any ordinary kernel-mode object (the kind that users can refer to with handles) that contains a data structure called a **dispatcher_header**, shown in Fig. 11-13. These objects include semaphores, mutexes, events, waitable timers, and other objects that threads can wait on to synchronize execution with other threads. They also include objects representing open files, processes, threads, and IPC ports. The dispatcher data structure contains a flag representing the signaled state of the object, and a queue of threads waiting for the object to be signaled.

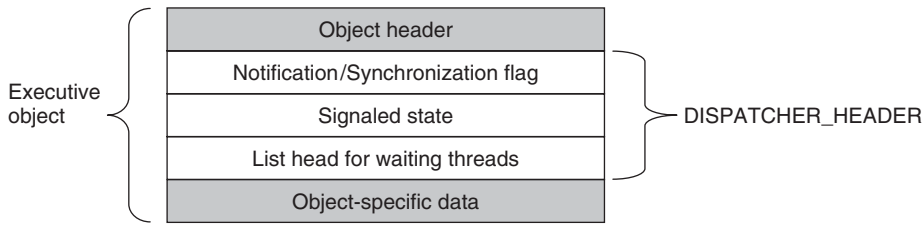

Figure 11-13. *dispatcher_header* data structure embedded in many executive objects (*dispatcher objects*).

Synchronization primitives, like semaphores, are natural dispatcher objects. Also timers, files, ports, threads, and processes use the dispatcher-object mechanisms for notifications. When a timer fires, I/O completes on a file, data are available on a port, or a thread or process terminates, the associated dispatcher object is signaled, waking all threads waiting for that event.

Since Windows uses a single unified mechanism for synchronization with kernel-mode objects, specialized APIs, such as wait3, for waiting for child processes in UNIX, are not needed to wait for events. Often threads want to wait for multiple events at once. In UNIX a process can wait for data to be available on any of 64 network sockets using the select system call. In Windows, there is a similar API **WaitForMultipleObjects**, but it allows for a thread to wait on any type of dispatcher object for which it has a handle. Up to 64 handles can be specified to WaitForMultipleObjects, as well as an optional timeout value. The thread becomes ready to run whenever any of the events associated with the handles is signaled or the timeout occurs.

There are actually two different procedures the kernel uses for making the threads waiting on a dispatcher object runnable. Signaling a **notification object** will make every waiting thread runnable. **Synchronization objects** make only the first waiting thread runnable and are used for dispatcher objects that implement

locking primitives, like mutexes. When a thread that is waiting for a lock begins running again, the first thing it does is to retry acquiring the lock. If only one thread can hold the lock at a time, all the other threads made runnable might immediately block, incurring lots of unnecessary context switching. The difference between dispatcher objects using synchronization vs. notification is a flag in the dispatcher_header structure.

As a little aside, mutexes in Windows are called "mutants" in the code because they were required to implement the OS/2 semantics of not automatically unlocking themselves when a thread holding one exited, something Cutler considered bizarre.

The Executive Layer

As shown in Fig. 11-11, below the kernel layer of NTOS there is the executive. The executive layer is written in C, is mostly architecture independent (the memory manager being a notable exception), and has been ported with only modest effort to new processors (MIPS, x86, PowerPC, Alpha, IA64, x64, and ARM). The executive contains a number of different components, all of which run using the control abstractions provided by the kernel layer.

Each component is divided into internal and external data structures and interfaces. The internal aspects of each component are hidden and used only within the component itself, while the external aspects are available to all the other components within the executive. A subset of the external interfaces are exported from the *ntoskrnl.exe* executable and device drivers can link to them as if the executive were a library. Microsoft calls many of the executive components "managers," because each is charge of managing some aspect of the operating services, such as I/O, memory, processes, objects, etc.

As with most operating systems, much of the functionality in the Windows executive is like library code, except that it runs in kernel mode so its data structures can be shared and protected from access by user-mode code, and so it can access kernel-mode state, such as the MMU control registers. But otherwise the executive is simply executing operating system functions on behalf of its caller, and thus runs in the thread of its called.

When any of the executive functions block waiting to synchronize with other threads, the user-mode thread is blocked, too. This makes sense when working on behalf of a particular user-mode thread, but it can be unfair when doing work related to common housekeeping tasks. To avoid hijacking the current thread when the executive determines that some housekeeping is needed, a number of kernel-mode threads are created when the system boots and dedicated to specific tasks, such as making sure that modified pages get written to disk.

For predictable, low-frequency tasks, there is a thread that runs once a second and has a laundry list of items to handle. For less predictable work there is the

pool of high-priority worker threads mentioned earlier which can be used to run bounded tasks by queuing a request and signaling the synchronization event that the worker threads are waiting on.

The **object manager** manages most of the interesting kernel-mode objects used in the executive layer. These include processes, threads, files, semaphores, I/O devices and drivers, timers, and many others. As described previously, kernel-mode objects are really just data structures allocated and used by the kernel. In Windows, kernel data structures have enough in common that it is very useful to manage many of them in a unified facility.

The facilities provided by the object manager include managing the allocation and freeing of memory for objects, quota accounting, supporting access to objects using handles, maintaining reference counts for kernel-mode pointer references as well as handle references, giving objects names in the NT namespace, and providing an extensible mechanism for managing the lifecycle for each object. Kernel data structures which need some of these facilities are managed by the object manager.

Object-manager objects each have a type which is used to specify exactly how the lifecycle of objects of that type is to be managed. These are not types in the object-oriented sense, but are simply a collection of parameters specified when the object type is created. To create a new type, an executive component calls an object-manager API to create a new type. Objects are so central to the functioning of Windows that the object manager will be discussed in more detail in the next section.

The **I/O manager** provides the framework for implementing I/O device drivers and provides a number of executive services specific to configuring, accessing, and performing operations on devices. In Windows, device drivers not only manage physical devices but they also provide extensibility to the operating system. Many functions that are compiled into the kernel on other systems are dynamically loaded and linked by the kernel on Windows, including network protocol stacks and file systems.

Recent versions of Windows have a lot more support for running device drivers in user mode, and this is the preferred model for new device drivers. There are hundreds of thousands of different device drivers for Windows working with more than a million distinct devices. This represents a lot of code to get correct. It is much better if bugs cause a device to become inaccessible by crashing in a user-mode process rather than causing the system to crash. Bugs in kernel-mode device drivers are the major source of the dreaded **BSOD** (**Blue Screen Of Death**) where Windows detects a fatal error within kernel mode and shuts down or reboots the system. BSOD's are comparable to kernel panics on UNIX systems.

In essence, Microsoft has now officially recognized what researchers in the area of microkernels such as MINIX 3 and L4 have known for years: the more code there is in the kernel, the more bugs there are in the kernel. Since device drivers make up something in the vicinity of 70% of the code in the kernel, the more

drivers that can be moved into user-mode processes, where a bug will only trigger the failure of a single driver (rather than bringing down the entire system), the better. The trend of moving code from the kernel to user-mode processes is expected to accelerate in the coming years.

The I/O manager also includes the plug-and-play and device power-management facilities. **Plug-and-play** comes into action when new devices are detected on the system. The plug-and-play subcomponent is first notified. It works with a service, the user-mode plug-and-play manager, to find the appropriate device driver and load it into the system. Getting the right one is not always easy and sometimes depends on sophisticated matching of the specific hardware device version to a particular version of the drivers. Sometimes a single device supports a standard interface which is supported by multiple different drivers, written by different companies.

We will study I/O further in Sec. 11.7 and the most important NT file system, NTFS, in Sec. 11.8.

Device power management reduces power consumption when possible, extending battery life on notebooks, and saving energy on desktops and servers. Getting power management correct can be challenging, as there are many subtle dependencies between devices and the buses that connect them to the CPU and memory. Power consumption is not affected just by what devices are powered-on, but also by the clock rate of the CPU, which is also controlled by the device power manager. We will take a more in depth look at power management in Sec. 11.9.

The **process manager** manages the creation and termination of processes and threads, including establishing the policies and parameters which govern them. But the operational aspects of threads are determined by the kernel layer, which controls scheduling and synchronization of threads, as well as their interaction with the control objects, like APCs. Processes contain threads, an address space, and a handle table containing the handles the process can use to refer to kernel-mode objects. Processes also include information needed by the scheduler for switching between address spaces and managing process-specific hardware information (such as segment descriptors). We will study process and thread management in Sec. 11.4.

The executive **memory manager** implements the demand-paged virtual memory architecture. It manages the mapping of virtual pages onto physical page frames, the management of the available physical frames, and management of the pagefile on disk used to back private instances of virtual pages that are no longer loaded in memory. The memory manager also provides special facilities for large server applications such as databases and programming language run-time components such as garbage collectors. We will study memory management later in this chapter, in Sec. 11.5.

The **cache manager** optimizes the performance of I/O to the file system by maintaining a cache of file-system pages in the kernel virtual address space. The cache manager uses virtually addressed caching, that is, organizing cached pages

in terms of their location in their files. This differs from physical block caching, as in UNIX, where the system maintains a cache of the physically addressed blocks of the raw disk volume.

Cache management is implemented using mapped files. The actual caching is performed by the memory manager. The cache manager need be concerned only with deciding what parts of what files to cache, ensuring that cached data is flushed to disk in a timely fashion, and managing the kernel virtual addresses used to map the cached file pages. If a page needed for I/O to a file is not available in the cache, the page will be faulted in using the memory manager. We will study the cache manager in Sec. 11.6.

The **security reference monitor** enforces Windows' elaborate security mechanisms, which support the international standards for computer security called **Common Criteria**, an evolution of United States Department of Defense Orange Book security requirements. These standards specify a large number of rules that a conforming system must meet, such as authenticated login, auditing, zeroing of allocated memory, and many more. One rules requires that all access checks be implemented by a single module within the system. In Windows, this module is the security reference monitor in the kernel. We will study the security system in more detail in Sec. 11.10.

The executive contains a number of other components that we will briefly describe. The **configuration manager** is the executive component which implements the registry, as described earlier. The registry contains configuration data for the system in file-system files called *hives*. The most critical hive is the *SYSTEM* hive which is loaded into memory at boot time. Only after the executive layer has successfully initialized its key components, including the I/O drivers that talk to the system disk, is the in-memory copy of the hive reassociated with the copy in the file system. Thus, if something bad happens while trying to boot the system, the on-disk copy is much less likely to be corrupted.

The LPC component provides for a highly efficient interprocess communication used between processes running on the same system. It is one of the data transports used by the standards-based remote procedure call facility to implement the client/server style of computing. RPC also uses named pipes and TCP/IP as transports.

LPC was substantially enhanced in Windows 8 (it is now called **ALPC**, for **Advanced LPC**) to provide support for new features in RPC, including RPC from kernel mode components, like drivers. LPC was a critical component in the original design of NT because it is used by the subsystem layer to implement communication between library stub routines that run in each process and the subsystem process which implements the facilities common to a particular operating system personality, such as Win32 or POSIX.

Windows 8 implemented a publish/subscibe service called **WNF** (**Windows Notification Facility**). WNF notifications are based on changes to an instance of WNF state data. A publisher declares an instance of state data (up to 4 KB) and

tells the operating system how long to maintain it (e.g., until the next reboot or permanently). A publisher atomically updates the state as appropriate. Subscribers can arrange to run code whenever an instance of state data is modified by a publisher. Because the WNF state instances contain a fixed amount of preallocated data, there is no queuing of data as in message-based IPC—with all the attendant resource-management problems. Subscribers are guaranteed only that they can see the latest version of a state instance.

This state-based approach gives WNF its principal advantage over other IPC mechanisms: publishers and subscribers are decoupled and can start and stop independently of each other. Publishers need not execute at boot time just to initialize their state instances, as those can be persisted by the operating system across reboots. Subscribers generally need not be concerned about past values of state instances when they start running, as all they should need to know about the state's history is encapsulated in the current state. In scenarios where past state values cannot be reasonably encapsulated, the current state can provide metadata for managing historical state, say, in a file or in a persisted section object used as a circular buffer. WNF is part of the native NT APIs and is not (yet) exposed via Win32 interfaces. But it is extensively used internally by the system to implement Win32 and WinRT APIs.

In Windows NT 4.0, much of the code related to the Win32 graphical interface was moved into the kernel because the then-current hardware could not provide the required performance. This code previously resided in the *csrss.exe* subsystem process which implemented the Win32 interfaces. The kernel-based GUI code resides in a special kernel-driver, *win32k.sys*. This change was expected to improve Win32 performance because the extra user-mode/kernel-mode transitions and the cost of switching address spaces to implement communication via LPC was eliminated. But it has not been as successful as expected because the requirements on code running in the kernel are very strict, and the additional overhead of running in kernel-mode offsets some of the gains from reducing switching costs.

The Device Drivers

The final part of Fig. 11-11 consists of the **device drivers**. Device drivers in Windows are dynamic link libraries which are loaded by the NTOS executive. Though they are primarily used to implement the drivers for specific hardware, such as physical devices and I/O buses, the device-driver mechanism is also used as the general extensibility mechanism for kernel mode. As described above, much of the Win32 subsystem is loaded as a driver.

The I/O manager organizes a data flow path for each instance of a device, as shown in Fig. 11-14. This path is called a **device stack** and consists of private instances of kernel device objects allocated for the path. Each device object in the device stack is linked to a particular driver object, which contains the table of

routines to use for the I/O request packets that flow through the device stack. In some cases the devices in the stack represent drivers whose sole purpose is to **filter** I/O operations aimed at a particular device, bus, or network driver. Filtering is used for a number of reasons. Sometimes preprocessing or postprocessing I/O operations results in a cleaner architecture, while other times it is just pragmatic because the sources or rights to modify a driver are not available and so filtering is used to work around the inability to modify those drivers. Filters can also implement completely new functionality, such as turning disks into partitions or multiple disks into RAID volumes.

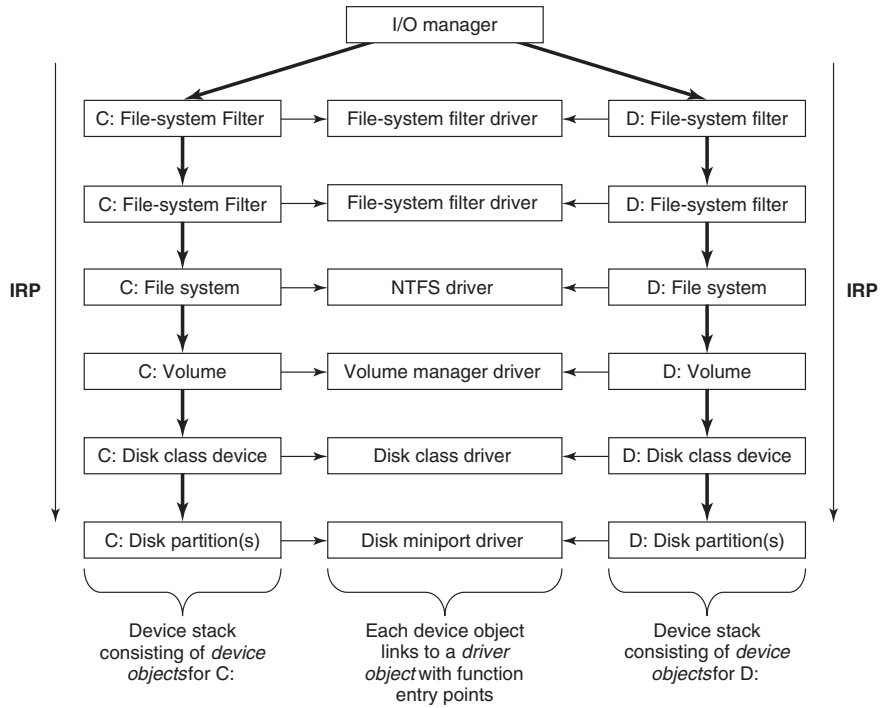

Figure 11-14. Simplified depiction of device stacks for two NTFS file volumes. The I/O request packet is passed from down the stack. The appropriate routines from the associated drivers are called at each level in the stack. The device stacks themselves consist of device objects allocated specifically to each stack.

The file systems are loaded as device drivers. Each instance of a volume for a file system has a device object created as part of the device stack for that volume. This device object will be linked to the driver object for the file system appropriate to the volume's formatting. Special filter drivers, called **file-system filter drivers**, can insert device objects before the file-system device object to apply functionality to the I/O requests being sent to each volume, such as inspecting data read or written for viruses.

The network protocols, such as Windows' integrated IPv4/IPv6 TCP/IP implementation, are also loaded as drivers using the I/O model. For compatibility with the older MS-DOS-based Windows, the TCP/IP driver implements a special protocol for talking to network interfaces on top of the Windows I/O model. There are other drivers that also implement such arrangements, which Windows calls **miniports**. The shared functionality is in a **class driver**. For example, common functionality for SCSI or IDE disks or USB devices is supplied by a class driver, which miniport drivers for each particular type of such devices link to as a library.

We will not discuss any particular device driver in this chapter, but will provide more detail about how the I/O manager interacts with device drivers in Sec. 11.7.

11.3.2 Booting Windows

Getting an operating system to run requires several steps. When a computer is turned on, the first processor is initialized by the hardware, and then set to start executing a program in memory. The only available code is in some form of nonvolatile CMOS memory that is initialized by the computer manufacturer (and sometimes updated by the user, in a process called **flashing**). Because the software persists in memory, and is only rarely updated, it is referred to as **firmware**. The firmware is loaded on PCs by the manufacturer of either the parentboard or the computer system. Historically PC firmware was a program called BIOS (Basic Input/Output System), but most new computers use **UEFI** (**Unified Extensible Firmware Interface**). UEFI improves over BIOS by supporting modern hardware, providing a more modular CPU-independent architecture, and supporting an extension model which simplifies booting over networks, provisioning new machines, and running diagnostics.

The main purpose of any firmware is to bring up the operating system by first loading small bootstrap programs found at the beginning of the disk-drive partitions. The Windows bootstrap programs know how to read enough information off a file-system volume or network to find the stand-alone Windows *BootMgr* program. *BootMgr* determines if the system had previously been hibernated or was in stand-by mode (special power-saving modes that allow the system to turn back on without restarting from the beginning of the bootstrap process). If so, *BootMgr* loads and executes *WinResume.exe*. Otherwise it loads and executes *WinLoad.exe* to perform a fresh boot. *WinLoad* loads the boot components of the system into memory: the kernel/executive (normally *ntoskrnl.exe*), the HAL (*hal.dll*), the file containing the SYSTEM hive, the *Win32k.sys* driver containing the kernel-mode parts of the Win32 subsystem, as well as images of any other drivers that are listed in the SYSTEM hive as **boot drivers**—meaning they are needed when the system first boots. If the system has Hyper-V enabled, WinLoad also loads and starts the hypervisor program.

Once the Windows boot components have been loaded into memory, control is handed over to the low-level code in NTOS which proceeds to initialize the HAL,

kernel, and executive layers, link in the driver images, and access/update configuration data in the SYSTEM hive. After all the kernel-mode components are initialized, the first user-mode process is created using for running the *smss.exe* program (which is like */etc/init* in UNIX systems).

Recent versions of Windows provide support for improving the security of the system at boot time. Many newer PCs contain a **TPM** (**Trusted Platform Module**), which is chip on the parentboard. chip is a secure cryptographic processor which protects secrets, such as encryption/decryption keys. The system's TPM can be used to protect system keys, such as those used by BitLocker to encrypt the disk. Protected keys are not revealed to the operating system until after TPM has verified that an attacker has not tampered with them. It can also provide other cryptographic functions, such as attesting to remote systems that the operating system on the local system had not been compromised.

The Windows boot programs have logic to deal with common problems users encounter when booting the system fails. Sometimes installation of a bad device driver, or running a program like *regedit* (which can corrupt the SYSTEM hive), will prevent the system from booting normally. There is support for ignoring recent changes and booting to the *last known good* configuration of the system. Other boot options include **safe-boot**, which turns off many optional drivers, and the **recovery console**, which fires up a *cmd.exe* command-line window, providing an experience similar to single-user mode in UNIX.

Another common problem for users has been that occasionally some Windows systems appear to be very flaky, with frequent (seemingly random) crashes of both the system and applications. Data taken from Microsoft's Online Crash Analysis program provided evidence that many of these crashes were due to bad physical memory, so the boot process in Windows provides the option of running an extensive memory diagnostic. Perhaps future PC hardware will commonly support ECC (or maybe parity) for memory, but most of the desktop, notebook, and handheld systems today are vulnerable to even single-bit errors in the tens of billions of memory bits they contain.

11.3.3 Implementation of the Object Manager

The object manager is probably the single most important component in the Windows executive, which is why we have already introduced many of its concepts. As described earlier, it provides a uniform and consistent interface for managing system resources and data structures, such as open files, processes, threads, memory sections, timers, devices, drivers, and semaphores. Even more specialized objects representing things like kernel transactions, profiles, security tokens, and Win32 desktops are managed by the object manager. Device objects link together the descriptions of the I/O system, including providing the link between the NT namespace and file-system volumes. The configuration manager uses an object of type **key** to link in the registry hives. The object manager itself has objects it uses

to manage the NT namespace and implement objects using a common facility. These are directory, symbolic link, and object-type objects.

The uniformity provided by the object manager has various facets. All these objects use the same mechanism for how they are created, destroyed, and accounted for in the quota system. They can all be accessed from user-mode processes using handles. There is a unified convention for managing pointer references to objects from within the kernel. Objects can be given names in the NT namespace (which is managed by the object manager). Dispatcher objects (objects that begin with the common data structure for signaling events) can use common synchronization and notification interfaces, like WaitForMultipleObjects. There is the common security system with ACLs enforced on objects opened by name, and access checks on each use of a handle. There are even facilities to help kernel-mode developers debug problems by tracing the use of objects.

A key to understanding objects is to realize that an (executive) object is just a data structure in the virtual memory accessible to kernel mode. These data structures are commonly used to represent more abstract concepts. As examples, executive file objects are created for each instance of a file-system file that has been opened. Process objects are created to represent each process.

A consequence of the fact that objects are just kernel data structures is that when the system is rebooted (or crashes) all objects are lost. When the system boots, there are no objects present at all, not even the object-type descriptors. All object types, and the objects themselves, have to be created dynamically by other components of the executive layer by calling the interfaces provided by the object manager. When objects are created and a name is specified, they can later be referenced through the NT namespace. So building up the objects as the system boots also builds the NT namespace.

Objects have a structure, as shown in Fig. 11-15. Each object contains a header with certain information common to all objects of all types. The fields in this header include the object's name, the object directory in which it lives in the NT namespace, and a pointer to a security descriptor representing the ACL for the object.

The memory allocated for objects comes from one of two heaps (or pools) of memory maintained by the executive layer. There are (malloc-like) utility functions in the executive that allow kernel-mode components to allocate either pageable or nonpageable kernel memory. Nonpageable memory is required for any data structure or kernel-mode object that might need to be accessed from a CPU priority level of 2 or more. This includes ISRs and DPCs (but not APCs) and the thread scheduler itself. The page-fault handler also requires its data structures to be allocated from nonpageable kernel memory to avoid recursion.

Most allocations from the kernel heap manager are achieved using per-processor lookaside lists which contain LIFO lists of allocations the same size. These LIFOs are optimized for lock-free operation, improving the performance and scalability of the system.

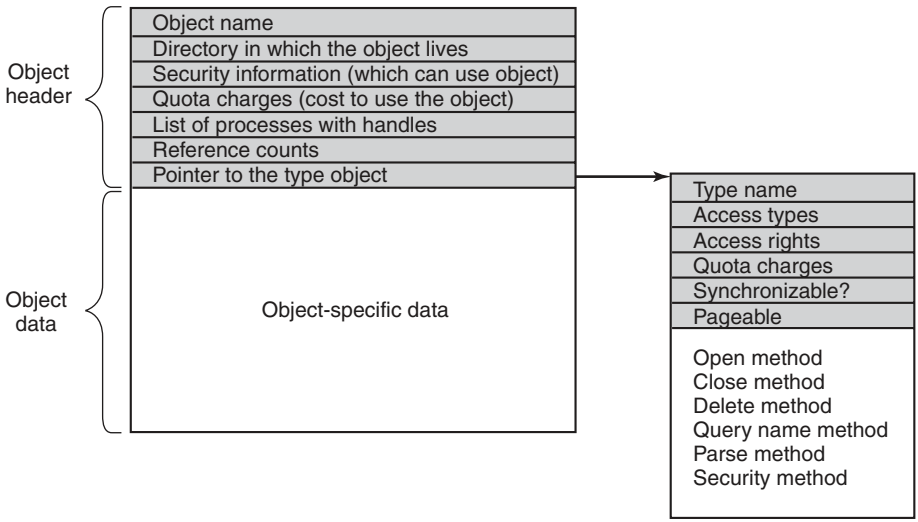

Figure 11-15. Structure of an executive object managed by the object manager

Each object header contains a quota-charge field, which is the charge levied against a process for opening the object. Quotas are used to keep a user from using too many system resources. There are separate limits for nonpageable kernel memory (which requires allocation of both physical memory and kernel virtual addresses) and pageable kernel memory (which uses up kernel virtual addresses). When the cumulative charges for either memory type hit the quota limit, allocations for that process fail due to insufficient resources. Quotas also are used by the memory manager to control working-set size, and by the thread manager to limit the rate of CPU usage.

Both physical memory and kernel virtual addresses are valuable resources. When an object is no longer needed, it should be removed and its memory and addresses reclaimed. But if an object is reclaimed while it is still in use, then the memory may be allocated to another object, and then the data structures are likely to become corrupted. It is easy for this to happen in the Windows executive layer because it is highly multithreaded, and implements many asynchronous operations (functions that return to their caller before completing work on the data structures passed to them).

To avoid freeing objects prematurely due to race conditions, the object manager implements a reference counting mechanism and the concept of a **referenced pointer**. A referenced pointer is needed to access an object whenever that object is in danger of being deleted. Depending on the conventions regarding each particular object type, there are only certain times when an object might be deleted by another thread. At other times the use of locks, dependencies between data structures, and even the fact that no other thread has a pointer to an object are sufficient to keep the object from being prematurely deleted.

Handles

User-mode references to kernel-mode objects cannot use pointers because they are too difficult to validate. Instead, kernel-mode objects must be named in some other way so the user code can refer to them. Windows uses **handles** to refer to kernel-mode objects. Handles are opaque values which are converted by the object manager into references to the specific kernel-mode data structure representing an object. Figure 11-16 shows the handle-table data structure used to translate handles into object pointers. The handle table is expandable by adding extra layers of indirection. Each process has its own table, including the system process which contains all the kernel threads not associated with a user-mode process.

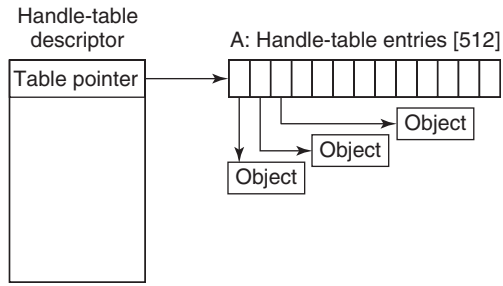

Figure 11-16. Handle table data structures for a minimal table using a single page for up to 512 handles.

Figure 11-17 shows a handle table with two extra levels of indirection, the maximum supported. It is sometimes convenient for code executing in kernel mode to be able to use handles rather than referenced pointers. These are called **kernel handles** and are specially encoded so that they can be distinguished from user-mode handles. Kernel handles are kept in the system processes' handle table and cannot be accessed from user mode. Just as most of the kernel virtual address space is shared across all processes, the system handle table is shared by all kernel components, no matter what the current user-mode process is.

Users can create new objects or open existing objects by making Win32 calls such as CreateSemaphore or OpenSemaphore. These are calls to library procedures that ultimately result in the appropriate system calls being made. The result of any successful call that creates or opens an object is a 64-bit handle-table entry that is stored in the process' private handle table in kernel memory. The 32-bit index of the handle's logical position in the table is returned to the user to use on subsequent calls. The 64-bit handle-table entry in the kernel contains two 32-bit words. One word contains a 29-bit pointer to the object's header. The low-order 3 bits are used as flags (e.g., whether the handle is inherited by processes it creates). These 3 bits are masked off before the pointer is followed. The other word contains a 32-bit rights mask. It is needed because permissions checking is done only

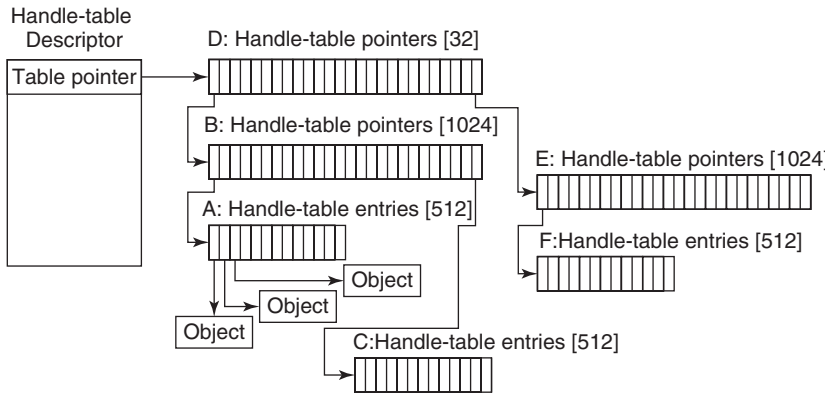

Figure 11-17. Handle-table data structures for a maximal table of up to 16 million handles.

at the time the object is created or opened. If a process has only read permission to an object, all the other rights bits in the mask will be 0s, giving the operating system the ability to reject any operation on the object other than reads.

The Object Namespace

Processes can share objects by having one process duplicate a handle to the object into the others. But this requires that the duplicating process have handles to the other processes, and is thus impractical in many situations, such as when the processes sharing an object are unrelated, or are protected from each other. In other cases it is important that objects persist even when they are not being used by any process, such as device objects representing physical devices, or mounted volumes, or the objects used to implement the object manager and the NT namespace itself. To address general sharing and persistence requirements, the object manager allows arbitrary objects to be given names in the NT namespace when they are created. However, it is up to the executive component that manipulates objects of a particular type to provide interfaces that support use of the object manager's naming facilities.

The NT namespace is hierarchical, with the object manager implementing directories and symbolic links. The namespace is also extensible, allowing any object type to specify extensions of the namespace by specifying a **Parse** routine. The *Parse* routine is one of the procedures that can be supplied for each object type when it is created, as shown in Fig. 11-18.

The *Open* procedure is rarely used because the default object-manager behavior is usually what is needed and so the procedure is specified as NULL for almost all object types.

Procedure	When called	Notes
Open	For every new handle	Rarely used
Parse	For object types that extend the namespace	Used for files and registry keys
Close	At last handle close	Clean up visible side effects
Delete	At last pointer dereference	Object is about to be deleted
Security	Get or set object's security descriptor	Protection
QueryName	Get object's name	Rarely used outside kernel

Figure 11-18. Object procedures supplied when specifying a new object type.

The *Close* and *Delete* procedures represent different phases of being done with an object. When the last handle for an object is closed, there may be actions necessary to clean up the state and these are performed by the *Close* procedure. When the final pointer reference is removed from the object, the *Delete* procedure is called so that the object can be prepared to be deleted and have its memory reused. With file objects, both of these procedures are implemented as callbacks into the I/O manager, which is the component that declared the file object type. The object-manager operations result in I/O operations that are sent down the device stack associated with the file object; the file system does most of the work.

The *Parse* procedure is used to open or create objects, like files and registry keys, that extend the NT namespace. When the object manager is attempting to open an object by name and encounters a leaf node in the part of the namespace it manages, it checks to see if the type for the leaf-node object has specified a *Parse* procedure. If so, it invokes the procedure, passing it any unused part of the path name. Again using file objects as an example, the leaf node is a device object representing a particular file-system volume. The *Parse* procedure is implemented by the I/O manager, and results in an I/O operation to the file system to fill in a file object to refer to an open instance of the file that the path name refers to on the volume. We will explore this particular example step-by-step below.

The *QueryName* procedure is used to look up the name associated with an object. The *Security* procedure is used to get, set, or delete the security descriptors on an object. For most object types this procedure is supplied as a standard entry point in the executive's security reference monitor component.

Note that the procedures in Fig. 11-18 do not perform the most useful operations for each type of object, such as read or write on files (or down and up on semaphores). Rather, the object manager procedures supply the functions needed to correctly set up access to objects and then clean up when the system is finished with them. The objects are made useful by the APIs that operate on the data structures the objects contain. System calls, like NtReadFile and NtWriteFile, use the process' handle table created by the object manager to translate a handle into a referenced pointer on the underlying object, such as a file object, which contains the data that is needed to implement the system calls.

Apart from the object-type callbacks, the object manager also provides a set of generic object routines for operations like creating objects and object types, duplicating handles, getting a referenced pointer from a handle or name, adding and subtracting reference counts to the object header, and NtClose (the generic function that closes all types of handles).

Although the object namespace is crucial to the entire operation of the system, few people know that it even exists because it is not visible to users without special viewing tools. One such viewing tool is *winobj*, available for free at the URL *www.microsoft.com/technet/sysinternals*. When run, this tool depicts an object namespace that typically contains the object directories listed in Fig. 11-19 as well as a few others.

Directory	Contents
\??	Starting place for looking up MS-DOS devices like C:
\DosDevices	Official name of \??, but really just a symbolic link to \??
\Device	All discovered I/O devices
\Driver	Objects corresponding to each loaded device driver
\ObjectTypes	The type objects such as those listed in Fig. 11-21
\Windows	Objects for sending messages to all the Win32 GUI windows
\BaseNamedObjects	User-created Win32 objects such as semaphores, mutexes, etc.
\Arcname	Partition names discovered by the boot loader
\NLS	National Language Support objects
\FileSystem	File-system driver objects and file system recognizer objects
\Security	Objects belonging to the security system
\KnownDLLs	Key shared libraries that are opened early and held open

Figure 11-19. Some typical directories in the object namespace.

The strangely named directory *\??* contains the names of all the MS-DOS-style device names, such as *A*: for the floppy disk and *C*: for the first hard disk. These names are actually symbolic links to the directory *\Device* where the device objects live. The name *\??* was chosen to make it alphabetically first so as to speed up lookup of all path names beginning with a drive letter. The contents of the other object directories should be self explanatory.

As described above, the object manager keeps a separate handle count in every object. This count is never larger than the referenced pointer count because each valid handle has a referenced pointer to the object in its handle-table entry. The reason for the separate handle count is that many types of objects may need to have their state cleaned up when the last user-mode reference disappears, even though they are not yet ready to have their memory deleted.

One example is file objects, which represent an instance of an opened file. In Windows, files can be opened for exclusive access. When the last handle for a file

object is closed it is important to delete the exclusive access at that point rather than wait for any incidental kernel references to eventually go away (e.g., after the last flush of data from memory). Otherwise closing and reopening a file from user mode may not work as expected because the file still appears to be in use.

Though the object manager has comprehensive mechanisms for managing object lifetimes within the kernel, neither the NT APIs nor the Win32 APIs provide a reference mechanism for dealing with the use of handles across multiple concurrent threads in user mode. Thus, many multithreaded applications have race conditions and bugs where they will close a handle in one thread before they are finished with it in another. Or they may close a handle multiple times, or close a handle that another thread is still using and reopen it to refer to a different object.

Perhaps the Windows APIs should have been designed to require a close API per object type rather than the single generic NtClose operation. That would have at least reduced the frequency of bugs due to user-mode threads closing the wrong handles. Another solution might be to embed a sequence field in each handle in addition to the index into the handle table.

To help application writers find problems like these in their programs, Windows has an **application verifier** that software developers can download from Microsoft. Similar to the verifier for drivers we will describe in Sec. 11.7, the application verifier does extensive rules checking to help programmers find bugs that might not be found by ordinary testing. It can also turn on a FIFO ordering for the handle free list, so that handles are not reused immediately (i.e., turns off the better-performing LIFO ordering normally used for handle tables). Keeping handles from being reused quickly transforms situations where an operation uses the wrong handle into use of a closed handle, which is easy to detect.

The device object is one of the most important and versatile kernel-mode objects in the executive. The type is specified by the I/O manager, which along with the device drivers, are the primary users of device objects. Device objects are closely related to drivers, and each device object usually has a link to a specific driver object, which describes how to access the I/O processing routines for the driver corresponding to the device.

Device objects represent hardware devices, interfaces, and buses, as well as logical disk partitions, disk volumes, and even file systems and kernel extensions like antivirus filters. Many device drivers are given names, so they can be accessed without having to open handles to instances of the devices, as in UNIX. We will use device objects to illustrate how the *Parse* procedure is used, as illustrated in Fig. 11-20:

1. When an executive component, such as the I/O manager implementing the native system call NtCreateFile, calls ObOpenObjectByName in the object manager, it passes a Unicode path name for the NT namespace, say *\??\C:\foo\bar*.

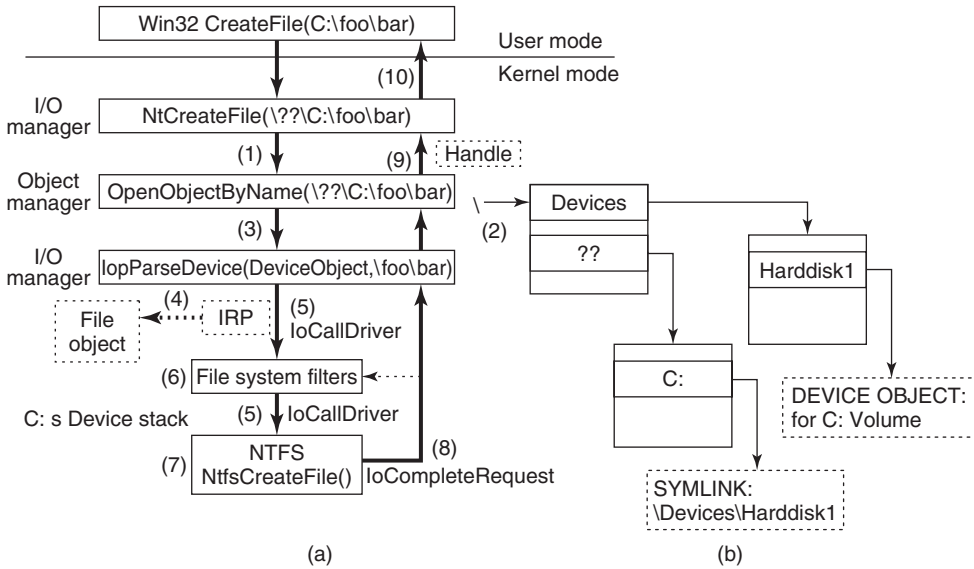

Figure 11-20. I/O and object manager steps for creating/opening a file and getting back a file handle.

2. The object manager searches through directories and symbolic links and ultimately finds that *\??\C:* refers to a device object (a type defined by the I/O manager). The device object is a leaf node in the part of the NT namespace that the object manager manages.

3. The object manager then calls the *Parse* procedure for this object type, which happens to be IopParseDevice implemented by the I/O manager. It passes not only a pointer to the device object it found (for *C:*), but also the remaining string *\foo\bar*.

4. The I/O manager will create an **IRP** (**I/O Request Packet**), allocate a file object, and send the request to the stack of I/O devices determined by the device object found by the object manager.

5. The IRP is passed down the I/O stack until it reaches a device object representing the file-system instance for *C:*. At each stage, control is passed to an entry point into the driver object associated with the device object at that level. The entry point used here is for CREATE operations, since the request is to create or open a file named *\foo\bar* on the volume.

6. The device objects encountered as the IRP heads toward the file system represent file-system filter drivers, which may modify the I/O operation before it reaches the file-system device object. Typically these intermediate devices represent system extensions like antivirus filters.

7. The file-system device object has a link to the file-system driver object, say NTFS. So, the driver object contains the address of the CREATE operation within NTFS.

8. NTFS will fill in the file object and return it to the I/O manager, which returns back up through all the devices on the stack until IopParseDevice returns to the object manager (see Sec. 11.8).

9. The object manager is finished with its namespace lookup. It received back an initialized object from the *Parse* routine (which happens to be a file object—not the original device object it found). So the object manager creates a handle for the file object in the handle table of the current process, and returns the handle to its caller.

10. The final step is to return back to the user-mode caller, which in this example is the Win32 API CreateFile, which will return the handle to the application.

Executive components can create new types dynamically, by calling the ObCreateObjectType interface to the object manager. There is no definitive list of object types and they change from release to release. Some of the more common ones in Windows are listed in Fig. 11-21. Let us briefly go over the object types in the figure.

Process and thread are obvious. There is one object for every process and every thread, which holds the main properties needed to manage the process or thread. The next three objects, semaphore, mutex, and event, all deal with interprocess synchronization. Semaphores and mutexes work as expected, but with various extra bells and whistles (e.g., maximum values and timeouts). Events can be in one of two states: signaled or nonsignaled. If a thread waits on an event that is in signaled state, the thread is released immediately. If the event is in nonsignaled state, it blocks until some other thread signals the event, which releases either all blocked threads (notification events) or just the first blocked thread (synchronization events). An event can also be set up so that after a signal has been successfully waited for, it will automatically revert to the nonsignaled state, rather than staying in the signaled state.

Port, timer, and queue objects also relate to communication and synchronization. Ports are channels between processes for exchanging LPC messages. Timers

Type	Description
Process	User process
Thread	Thread within a process
Semaphore	Counting semaphore used for interprocess synchronization
Mutex	Binary semaphore used to enter a critical region
Event	Synchronization object with persistent state (signaled/not)
ALPC port	Mechanism for interprocess message passing
Timer	Object allowing a thread to sleep for a fixed time interval
Queue	Object used for completion notification on asynchronous I/O
Open file	Object associated with an open file
Access token	Security descriptor for some object
Profile	Data structure used for profiling CPU usage
Section	Object used for representing mappable files
Key	Registry key, used to attach registry to object-manager namespace
Object directory	Directory for grouping objects within the object manager
Symbolic link	Refers to another object manager object by path name
Device	I/O device object for a physical device, bus, driver, or volume instance
Device driver	Each loaded device driver has its own object

Figure 11-21. Some common executive object types managed by the object manager.

provide a way to block for a specific time interval. Queues (known internally as **KQUEUES**) are used to notify threads that a previously started asynchronous I/O operation has completed or that a port has a message waiting. Queues are designed to manage the level of concurrency in an application, and are also used in high-performance multiprocessor applications, like SQL.

Open file objects are created when a file is opened. Files that are not opened do not have objects managed by the object manager. Access tokens are security objects. They identify a user and tell what special privileges the user has, if any. Profiles are structures used for storing periodic samples of the program counter of a running thread to see where the program is spending its time.

Sections are used to represent memory objects that applications can ask the memory manager to map into their address space. They record the section of the file (or page file) that represents the pages of the memory object when they are on disk. Keys represent the mount point for the registry namespace on the object manager namespace. There is usually only one key object, named *REGISTRY*, which connects the names of the registry keys and values to the NT namespace.

Object directories and symbolic links are entirely local to the part of the NT namespace managed by the object manager. They are similar to their file system counterparts: directories allow related objects to be collected together. Symbolic

links allow a name in one part of the object namespace to refer to an object in a different part of the object namespace.

Each device known to the operating system has one or more device objects that contain information about it and are used to refer to the device by the system. Finally, each device driver that has been loaded has a driver object in the object space. The driver objects are shared by all the device objects that represent instances of the devices controlled by those drivers.

Other objects (not shown) have more specialized purposes, such as interacting with kernel transactions, or the Win32 thread pool's worker thread factory.

11.3.4 Subsystems, DLLs, and User-Mode Services

Going back to Fig. 11-4, we see that the Windows operating system consists of components in kernel mode and components in user mode. We have now completed our overview of the kernel-mode components; so it is time to look at the user-mode components, of which three kinds are particularly important to Windows: environment subsystems, DLLs, and service processes.

We have already described the Windows subsystem model; we will not go into more detail now other than to mention that in the original design of NT, subsystems were seen as a way of supporting multiple operating system personalities with the same underlying software running in kernel mode. Perhaps this was an attempt to avoid having operating systems compete for the same platform, as VMS and Berkeley UNIX did on DEC's VAX. Or maybe it was just that nobody at Microsoft knew whether OS/2 would be a success as a programming interface, so they were hedging their bets. In any case, OS/2 became irrelevant, and a latecomer, the Win32 API designed to be shared with Windows 95, became dominant.

A second key aspect of the user-mode design of Windows is the dynamic link library (DLL) which is code that is linked to executable programs at run time rather than compile time. Shared libraries are not a new concept, and most modern operating systems use them. In Windows, almost all libraries are DLLs, from the system library *ntdll.dll* that is loaded into every process to the high-level libraries of common functions that are intended to allow rampant code-reuse by application developers.

DLLs improve the efficiency of the system by allowing common code to be shared among processes, reduce program load times from disk by keeping commonly used code around in memory, and increase the serviceability of the system by allowing operating system library code to be updated without having to recompile or relink all the application programs that use it.

On the other hand, shared libraries introduce the problem of versioning and increase the complexity of the system because changes introduced into a shared library to help one particular program have the potential of exposing latent bugs in other applications, or just breaking them due to changes in the implementation—a problem that in the Windows world is referred to as **DLL hell**.

The implementation of DLLs is simple in concept. Instead of the compiler emitting code that calls directly to subroutines in the same executable image, a level of indirection is introduced: the **IAT** (**Import Address Table**). When an executable is loaded it is searched for the list of DLLs that must also be loaded (this will be a graph in general, as the listed DLLs will themselves will generally list other DLLs needed in order to run). The required DLLs are loaded and the IAT is filled in for them all.

The reality is more complicated. Another problem is that the graphs that represent the relationships between DLLs can contain cycles, or have nondeterministic behaviors, so computing the list of DLLs to load can result in a sequence that does not work. Also, in Windows the DLL libraries are given a chance to run code whenever they are loaded into a process, or when a new thread is created. Generally, this is so they can perform initialization, or allocate per-thread storage, but many DLLs perform a lot of computation in these *attach* routines. If any of the functions called in an *attach* routine needs to examine the list of loaded DLLs, a deadlock can occur, hanging the process.

DLLs are used for more than just sharing common code. They enable a *hosting* model for extending applications. Internet Explorer can download and link to DLLs called **ActiveX controls**. At the other end of the Internet, Web servers also load dynamic code to produce a better Web experience for the pages they display. Applications like Microsoft *Office* link and run DLLs to allow *Office* to be used as a platform for building other applications. The COM (component object model) style of programming allows programs to dynamically find and load code written to provide a particular published interface, which leads to in-process hosting of DLLs by almost all the applications that use COM.

All this dynamic loading of code has resulted in even greater complexity for the operating system, as library version management is not just a matter of matching executables to the right versions of the DLLs, but sometimes loading multiple versions of the same DLL into a process—which Microsoft calls **side-by-side**. A single program can host two different dynamic code libraries, each of which may want to load the same Windows library—yet have different version requirements for that library.

A better solution would be hosting code in separate processes. But out-of--process hosting of code results has lower performance, and makes for a more complicated programming model in many cases. Microsoft has yet to develop a good solution for all of this complexity in user mode. It makes one yearn for the relative simplicity of kernel mode.

One of the reasons that kernel mode has less complexity than user mode is that it supports relatively few extensibility opportunities outside of the device-driver model. In Windows, system functionality is extended by writing user-mode services. This worked well enough for subsystems, and works even better when only a few new services are being provided rather than a complete operating system personality. There are few functional differences between services implemented in the

kernel and services implemented in user-mode processes. Both the kernel and process provide private address spaces where data structures can be protected and service requests can be scrutinized.

However, there can be significant performance differences between services in the kernel vs. services in user-mode processes. Entering the kernel from user mode is slow on modern hardware, but not as slow as having to do it twice because you are switching back and forth to another process. Also cross-process communication has lower bandwidth.

Kernel-mode code can (carefully) access data at the user-mode addresses passed as parameters to its system calls. With user-mode services, either those data must be copied to the service process, or some games be played by mapping memory back and forth (the ALPC facilities in Windows handle this under the covers).

In the future it is possible that the hardware costs of crossing between address spaces and protection modes will be reduced, or perhaps even become irrelevant. The Singularity project in Microsoft Research (Fandrich et al., 2006) uses run-time techniques, like those used with C# and Java, to make protection a completely software issue. No hardware switching between address spaces or protection modes is required.

Windows makes significant use of user-mode service processes to extend the functionality of the system. Some of these services are strongly tied to the operation of kernel-mode components, such as *lsass.exe* which is the local security authentication service which manages the token objects that represent user-identity, as well as managing encryption keys used by the file system. The user-mode plug-and-play manager is responsible for determining the correct driver to use when a new hardware device is encountered, installing it, and telling the kernel to load it. Many facilities provided by third parties, such as antivirus and digital rights management, are implemented as a combination of kernel-mode drivers and user-mode services.

The Windows *taskmgr.exe* has a tab which identifies the services running on the system. Multiple services can be seen to be running in the same process (**svchost.exe**). Windows does this for many of its own boot-time services to reduce the time needed to start up the system. Services can be combined into the same process as long as they can safely operate with the same security credentials.

Within each of the shared service processes, individual services are loaded as DLLs. They normally share a pool of threads using the Win32 thread-pool facility, so that only the minimal number of threads needs to be running across all the resident services.

Services are common sources of security vulnerabilities in the system because they are often accessible remotely (depending on the TCP/IP firewall and IP Security settings), and not all programmers who write services are as careful as they should be to validate the parameters and buffers that are passed in via RPC.

The number of services running constantly in Windows is staggering. Yet few of those services ever receive a single request, though if they do it is likely to be

from an attacker attempting to exploit a vulnerability. As a result more and more services in Windows are turned off by default, particularly on versions of Windows Server.

11.4 PROCESSES AND THREADS IN WINDOWS

Windows has a number of concepts for managing the CPU and grouping resources together. In the following sections we will examine these, discussing some of the relevant Win32 API calls, and show how they are implemented.

11.4.1 Fundamental Concepts

In Windows processes are containers for programs. They hold the virtual address space, the handles that refer to kernel-mode objects, and threads. In their role as a container for threads they hold common resources used for thread execution, such as the pointer to the quota structure, the shared token object, and default parameters used to initialize threads—including the priority and scheduling class. Each process has user-mode system data, called the **PEB** (**Process Environment Block**). The PEB includes the list of loaded modules (i.e., the EXE and DLLs), the memory containing environment strings, the current working directory, and data for managing the process' heaps—as well as lots of special-case Win32 cruft that has been added over time.

Threads are the kernel's abstraction for scheduling the CPU in Windows. Priorities are assigned to each thread based on the priority value in the containing process. Threads can also be **affinitized** to run only on certain processors. This helps concurrent programs running on multicore chips or multiprocessors to explicitly spread out work. Each thread has two separate call stacks, one for execution in user mode and one for kernel mode. There is also a **TEB** (**Thread Environment Block**) that keeps user-mode data specific to the thread, including per-thread storage (**Thread Local Storage**) and fields for Win32, language and cultural localization, and other specialized fields that have been added by various facilities.

Besides the PEBs and TEBs, there is another data structure that kernel mode shares with each process, namely, **user shared data**. This is a page that is writable by the kernel, but read-only in every user-mode process. It contains a number of values maintained by the kernel, such as various forms of time, version information, amount of physical memory, and a large number of shared flags used by various user-mode components, such as COM, terminal services, and the debuggers. The use of this read-only shared page is purely a performance optimization, as the values could also be obtained by a system call into kernel mode. But system calls are much more expensive than a single memory access, so for some system-maintained fields, such as the time, this makes a lot of sense. The other fields, such as the current time zone, change infrequently (except on airborne computers),

but code that relies on these fields must query them often just to see if they have changed. As with many performance hacks, it is a bit ugly, but it works.

Processes

Processes are created from section objects, each of which describes a memory object backed by a file on disk. When a process is created, the creating process receives a handle that allows it to modify the new process by mapping sections, allocating virtual memory, writing parameters and environmental data, duplicating file descriptors into its handle table, and creating threads. This is very different than how processes are created in UNIX and reflects the difference in the target systems for the original designs of UNIX vs. Windows.

As described in Sec. 11.1, UNIX was designed for 16-bit single-processor systems that used swapping to share memory among processes. In such systems, having the process as the unit of concurrency and using an operation like fork to create processes was a brilliant idea. To run a new process with small memory and no virtual memory hardware, processes in memory have to be swapped out to disk to create space. UNIX originally implemented fork simply by swapping out the parent process and handing its physical memory to the child. The operation was almost free.

In contrast, the hardware environment at the time Cutler's team wrote NT was 32-bit multiprocessor systems with virtual memory hardware to share 1–16 MB of physical memory. Multiprocessors provide the opportunity to run parts of programs concurrently, so NT used processes as containers for sharing memory and object resources, and used threads as the unit of concurrency for scheduling.

Of course, the systems of the next few years will look nothing like either of these target environments, having 64-bit address spaces with dozens (or hundreds) of CPU cores per chip socket and dozens or hundreds gigabytes of physical memory. This memory may be radically different from current RAM as well. Current RAM loses its contents when powered off, but **phase-change memories** now in the pipeline keep their values (like disks) even when powered off. Also expect **flash devices** to replace hard disks, broader support for virtualization, ubiquitous networking, and support for synchronization innovations like **transactional memory**. Windows and UNIX will continue to be adapted to new hardware realities, but what will be really interesting is to see what new operating systems are designed specifically for systems based on these advances.

Jobs and Fibers

Windows can group processes together into jobs. Jobs group processes in order to apply constraints to them and the threads they contain, such as limiting resource use via a shared quota or enforcing a **restricted token** that prevents threads from accessing many system objects. The most significant property of jobs for

resource management is that once a process is in a job, all processes' threads in those processes create will also be in the job. There is no escape. As suggested by the name, jobs were designed for situations that are more like batch processing than ordinary interactive computing.

In Modern Windows, jobs are used to group together the processes that are executing a modern application. The processes that comprise a running application need to be identified to the operating system so it can manage the entire application on behalf of the user.

Figure 11-22 shows the relationship between jobs, processes, threads, and fibers. Jobs contain processes. Processes contain threads. But threads do not contain fibers. The relationship of threads to fibers is normally many-to-many.

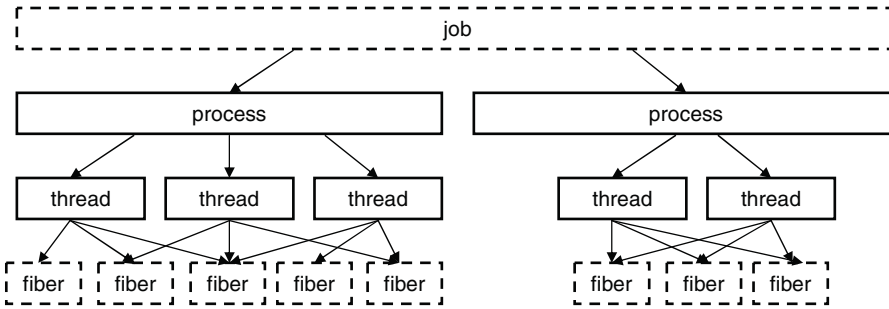

Figure 11-22. The relationship between jobs, processes, threads, and fibers. Jobs and fibers are optional; not all processes are in jobs or contain fibers.

Fibers are created by allocating a stack and a user-mode fiber data structure for storing registers and data associated with the fiber. Threads are converted to fibers, but fibers can also be created independently of threads. Such a fiber will not run until a fiber already running on a thread explicitly calls SwitchToFiber to run the fiber. Threads could attempt to switch to a fiber that is already running, so the programmer must provide synchronization to prevent this.

The primary advantage of fibers is that the overhead of switching between fibers is much lower than switching between threads. A thread switch requires entering and exiting the kernel. A fiber switch saves and restores a few registers without changing modes at all.

Although fibers are cooperatively scheduled, if there are multiple threads scheduling the fibers, a lot of careful synchronization is required to make sure fibers do not interfere with each other. To simplify the interaction between threads and fibers, it is often useful to create only as many threads as there are processors to run them, and affinitize the threads to each run only on a distinct set of available processors, or even just one processor.

Each thread can then run a particular subset of the fibers, establishing a one-to-many relationship between threads and fibers which simplifies synchronization. Even so there are still many difficulties with fibers. Most of the Win32 libraries

are completely unaware of fibers, and applications that attempt to use fibers as if they were threads will encounter various failures. The kernel has no knowledge of fibers, and when a fiber enters the kernel, the thread it is executing on may block and the kernel will schedule an arbitrary thread on the processor, making it unavailable to run other fibers. For these reasons fibers are rarely used except when porting code from other systems that explicitly need the functionality provided by fibers.

Thread Pools and User-Mode Scheduling

The Win32 **thread pool** is a facility that builds on top of the Windows thread model to provide a better abstraction for certain types of programs. Thread creation is too expensive to be invoked every time a program wants to execute a small task concurrently with other tasks in order to take advantage of multiple processors. Tasks can be grouped together into larger tasks but this reduces the amount of exploitable concurrency in the program. An alternative approach is for a program to allocate a limited number of threads, and maintain a queue of tasks that need to be run. As a thread finishes the execution of a task, it takes another one from the queue. This model separates the resource-management issues (how many processors are available and how many threads should be created) from the programming model (what is a task and how are tasks synchronized). Windows formalizes this solution into the Win32 thread pool, a set of APIs for automatically managing a dynamic pool of threads and dispatching tasks to them.

Thread pools are not a perfect solution, because when a thread blocks for some resource in the middle of a task, the thread cannot switch to a different task. Thus, the thread pool will inevitably create more threads than there are processors available, so if runnable threads are available to be scheduled even when other threads have blocked. The thread pool is integrated with many of the common synchronization mechanisms, such as awaiting the completion of I/O or blocking until a kernel event is signaled. Synchronization can be used as triggers for queuing a task so threads are not assigned the task before it is ready to run.

The implementation of the thread pool uses the same queue facility provided for synchronization with I/O completion, together with a kernel-mode thread factory which adds more threads to the process as needed to keep the available number of processors busy. Small tasks exist in many applications, but particularly in those that provide services in the client/server model of computing, where a stream of requests are sent from the clients to the server. Use of a thread pool for these scenarios improves the efficiency of the system by reducing the overhead of creating threads and moving the decisions about how to manage the threads in the pool out of the application and into the operating system.

What programmers see as a single Windows thread is actually two threads: one that runs in kernel mode and one that runs in user mode. This is precisely the same

model that UNIX has. Each of these threads is allocated its own stack and its own memory to save its registers when not running. The two threads appear to be a single thread because they do not run at the same time. The user thread operates as an extension of the kernel thread, running only when the kernel thread switches to it by returning from kernel mode to user mode. When a user thread wants to perform a system call, encounters a page fault, or is preempted, the system enters kernel mode and switches back to the corresponding kernel thread. It is normally not possible to switch between user threads without first switching to the corresponding kernel thread, switching to the new kernel thread, and then switching to its user thread.

Most of the time the difference between user and kernel threads is transparent to the programmer. However, in Windows 7 Microsoft added a facility called **UMS** (**User-Mode Scheduling**), which exposes the distinction. UMS is similar to facilities used in other operating systems, such as **scheduler activations**. It can be used to switch between user threads without first having to enter the kernel, providing the benefits of fibers, but with much better integration into Win32—since it uses real Win32 threads.

The implementation of UMS has three key elements:

1. *User-mode switching*: a user-mode scheduler can be written to switch between user threads without entering the kernel. When a user thread does enter kernel mode, UMS will find the corresponding kernel thread and immediately switch to it.
2. *Reentering the user-mode scheduler*: when the execution of a kernel thread blocks to await the availability of a resource, UMS switches to a special user thread and executes the user-mode scheduler so that a different user thread can be scheduled to run on the current processor. This allows the current process to continue using the current processor for its full turn rather than having to get in line behind other processes when one of its threads blocks.
3. *System-call completion*: after a blocked kernel thread eventually is finished, a notification containing the results of the system calls is queued for the user-mode scheduler so that it can switch to the corresponding user thread next time it makes a scheduling decision.

UMS does not include a user-mode scheduler as part of Windows. UMS is intended as a low-level facility for use by run-time libraries used by programming-language and server applications to implement lightweight threading models that do not conflict with kernel-level thread scheduling. These run-time libraries will normally implement a user-mode scheduler best suited to their environment. A summary of these abstractions is given in Fig. 11-23.

Name	Description	Notes
Job	Collection of processes that share quotas and limits	Used in AppContainers
Process	Container for holding resources	
Thread	Entity scheduled by the kernel	
Fiber	Lightweight thread managed entirely in user space	Rarely used
Thread pool	Task-oriented programming model	Built on top of threads
User-mode thread	Abstraction allowing user-mode thread switching	An extension of threads

Figure 11-23. Basic concepts used for CPU and resource management.

Threads

Every process normally starts out with one thread, but new ones can be created dynamically. Threads form the basis of CPU scheduling, as the operating system always selects a thread to run, not a process. Consequently, every thread has a state (ready, running, blocked, etc), whereas processes do not have scheduling states. Threads can be created dynamically by a Win32 call that specifies the address within the enclosing process' address space at which it is to start running.

Every thread has a thread ID, which is taken from the same space as the process IDs, so a single ID can never be in use for both a process and a thread at the same time. Process and thread IDs are multiples of four because they are actually allocated by the executive using a special handle table set aside for allocating IDs. The system is reusing the scalable handle-management facility shown in Figs. 11-16 and 11-17. The handle table does not have references on objects, but does use the pointer field to point at the process or thread so that the lookup of a process or thread by ID is very efficient. FIFO ordering of the list of free handles is turned on for the ID table in recent versions of Windows so that IDs are not immediately reused. The problems with immediate reuse are explored in the problems at the end of this chapter.

A thread normally runs in user mode, but when it makes a system call it switches to kernel mode and continues to run as the same thread with the same properties and limits it had in user mode. Each thread has two stacks, one for use when it is in user mode and one for use when it is in kernel mode. Whenever a thread enters the kernel, it switches to the kernel-mode stack. The values of the user-mode registers are saved in a **CONTEXT** data structure at the base of the kernel-mode stack. Since the only way for a user-mode thread to not be running is for it to enter the kernel, the CONTEXT for a thread always contains its register state when it is not running. The CONTEXT for each thread can be examined and modified from any process with a handle to the thread.

Threads normally run using the access token of their containing process, but in certain cases related to client/server computing, a thread running in a service process can impersonate its client, using a temporary access token based on the client's

token so it can perform operations on the client's behalf. (In general a service cannot use the client's actual token, as the client and server may be running on different systems.)

Threads are also the normal focal point for I/O. Threads block when performing synchronous I/O, and the outstanding I/O request packets for asynchronous I/O are linked to the thread. When a thread is finished executing, it can exit. Any I/O requests pending for the thread will be canceled. When the last thread still active in a process exits, the process terminates.

It is important to realize that threads are a scheduling concept, not a resource-ownership concept. Any thread is able to access all the objects that belong to its process. All it has to do is use the handle value and make the appropriate Win32 call. There is no restriction on a thread that it cannot access an object because a different thread created or opened it. The system does not even keep track of which thread created which object. Once an object handle has been put in a process' handle table, any thread in the process can use it, even it if is impersonating a different user.

As described previously, in addition to the normal threads that run within user processes Windows has a number of system threads that run only in kernel mode and are not associated with any user process. All such system threads run in a special process called the **system process**. This process does not have a user-mode address space. It provides the environment that threads execute in when they are not operating on behalf of a specific user-mode process. We will study some of these threads later when we come to memory management. Some perform administrative tasks, such as writing dirty pages to the disk, while others form the pool of worker threads that are assigned to run specific short-term tasks delegated by executive components or drivers that need to get some work done in the system process.

11.4.2 Job, Process, Thread, and Fiber Management API Calls

New processes are created using the Win32 API function CreateProcess. This function has many parameters and lots of options. It takes the name of the file to be executed, the command-line strings (unparsed), and a pointer to the environment strings. There are also flags and values that control many details such as how security is configured for the process and first thread, debugger configuration, and scheduling priorities. A flag also specifies whether open handles in the creator are to be passed to the new process. The function also takes the current working directory for the new process and an optional data structure with information about the GUI Window the process is to use. Rather than returning just a process ID for the new process, Win32 returns both handles and IDs, both for the new process and for its initial thread.

The large number of parameters reveals a number of differences from the design of process creation in UNIX.

1. The actual search path for finding the program to execute is buried in the library code for Win32, but managed more explicitly in UNIX.
2. The current working directory is a kernel-mode concept in UNIX but a user-mode string in Windows. Windows *does* open a handle on the current directory for each process, with the same annoying effect as in UNIX: you cannot delete the directory, unless it happens to be across the network, in which case you *can* delete it.
3. UNIX parses the command line and passes an array of parameters, while Win32 leaves argument parsing up to the individual program. As a consequence, different programs may handle wildcards (e.g., *.txt) and other special symbols in an inconsistent way.
4. Whether file descriptors can be inherited in UNIX is a property of the handle. In Windows it is a property of both the handle and a parameter to process creation.
5. Win32 is GUI oriented, so new processes are directly passed information about their primary window, while this information is passed as parameters to GUI applications in UNIX.
6. Windows does not have a SETUID bit as a property of the executable, but one process can create a process that runs as a different user, as long as it can obtain a token with that user's credentials.
7. The process and thread handle returned from Windows can be used at any time to modify the new process/thread in many substantive ways, including modifying the virtual memory, injecting threads into the process, and altering the execution of threads. UNIX makes modifications to the new process only between the fork and exec calls, and only in limited ways as exec throws out all the user-mode state of the process.

Some of these differences are historical and philosophical. UNIX was designed to be command-line oriented rather than GUI oriented like Windows. UNIX users are more sophisticated, and they understand concepts like *PATH* variables. Windows inherited a lot of legacy from MS-DOS.

The comparison is also skewed because Win32 is a user-mode wrapper around the native NT process execution, much as the *system* library function wraps fork/exec in UNIX. The actual NT system calls for creating processes and threads, NtCreateProcess and NtCreateThread, are simpler than the Win32 versions. The main parameters to NT process creation are a handle on a section representing the program file to run, a flag specifying whether the new process should, by default, inherit handles from the creator, and parameters related to the security model. All the details of setting up the environment strings and creating the initial thread are

left to user-mode code that can use the handle on the new process to manipulate its virtual address space directly.

To support the POSIX subsystem, native process creation has an option to create a new process by copying the virtual address space of another process rather than mapping a section object for a new program. This is used only to implement fork for POSIX, and not by Win32. Since POSIX no longer ships with Windows, process duplication has little use—though sometimes enterprising developers come up with special uses, similar to uses of fork without exec in UNIX.

Thread creation passes the CPU context to use for the new thread (which includes the stack pointer and initial instruction pointer), a template for the TEB, and a flag saying whether the thread should be immediately run or created in a suspended state (waiting for somebody to call NtResumeThread on its handle). Creation of the user-mode stack and pushing of the *argv*/*argc* parameters is left to user-mode code calling the native NT memory-management APIs on the process handle.

In the Windows Vista release, a new native API for processes, NtCreateUserProcess, was added which moves many of the user-mode steps into the kernel-mode executive, and combines process creation with creation of the initial thread. The reason for the change was to support the use of processes as security boundaries. Normally, all processes created by a user are considered to be equally trusted. It is the user, as represented by a token, that determines where the trust boundary is. NtCreateUserProcess allows processes to also provide trust boundaries, but this means that the creating process does not have sufficient rights regarding a new process handle to implement the details of process creation in user mode for processes that are in a different trust environment. The primary use of a process in a different trust boundary (called **protected processes**) is to support forms of digital rights management, which protect copyrighted material from being used improperly. Of course, protected processes only target user-mode attacks against protected content and cannot prevent kernel-mode attacks.

Interprocess Communication

Threads can communicate in a wide variety of ways, including pipes, named pipes, mailslots, sockets, remote procedure calls, and shared files. Pipes have two modes: byte and message, selected at creation time. Byte-mode pipes work the same way as in UNIX. Message-mode pipes are somewhat similar but preserve message boundaries, so that four writes of 128 bytes will be read as four 128-byte messages, and not as one 512-byte message, as might happen with byte-mode pipes. Named pipes also exist and have the same two modes as regular pipes. Named pipes can also be used over a network but regular pipes cannot.

Mailslots are a feature of the now-defunct OS/2 operating system implemented in Windows for compatibility. They are similar to pipes in some ways, but not all. For one thing, they are one way, whereas pipes are two way. They could

be used over a network but do not provide guaranteed delivery. Finally, they allow the sending process to broadcast a message to many receivers, instead of to just one receiver. Both mailslots and named pipes are implemented as file systems in Windows, rather than executive functions. This allows them to be accessed over the network using the existing remote file-system protocols.

Sockets are like pipes, except that they normally connect processes on different machines. For example, one process writes to a socket and another one on a remote machine reads from it. Sockets can also be used to connect processes on the same machine, but since they entail more overhead than pipes, they are generally only used in a networking context. Sockets were originally designed for Berkeley UNIX, and the implementation was made widely available. Some of the Berkeley code and data structures are still present in Windows today, as acknowledged in the release notes for the system.

RPCs are a way for process *A* to have process *B* call a procedure in *B*'s address space on *A*'s behalf and return the result to *A*. Various restrictions on the parameters exist. For example, it makes no sense to pass a pointer to a different process, so data structures have to be packaged up and transmitted in a nonprocess-specific way. RPC is normally implemented as an abstraction layer on top of a transport layer. In the case of Windows, the transport can be TCP/IP sockets, named pipes, or ALPC. ALPC (Advanced Local Procedure Call) is a message-passing facility in the kernel-mode executive. It is optimized for communicating between processes on the local machine and does not operate across the network. The basic design is for sending messages that generate replies, implementing a lightweight version of remote procedure call which the RPC package can build on top of to provide a richer set of features than available in ALPC. ALPC is implemented using a combination of copying parameters and temporary allocation of shared memory, based on the size of the messages.

Finally, processes can share objects. This includes section objects, which can be mapped into the virtual address space of different processes at the same time. All writes done by one process then appear in the address spaces of the other processes. Using this mechanism, the shared buffer used in producer-consumer problems can easily be implemented.

Synchronization

Processes can also use various types of synchronization objects. Just as Windows provides numerous interprocess communication mechanisms, it also provides numerous synchronization mechanisms, including semaphores, mutexes, critical regions, and events. All of these mechanisms work with threads, not processes, so that when a thread blocks on a semaphore, other threads in that process (if any) are not affected and can continue to run.

A semaphore can be created using the CreateSemaphore Win32 API function, which can also initialize it to a given value and define a maximum value as well.

Semaphores are kernel-mode objects and thus have security descriptors and handles. The handle for a semaphore can be duplicated using DuplicateHandle and passed to another process so that multiple processes can synchronize on the same semaphore. A semaphore can also be given a name in the Win32 namespace and have an ACL set to protect it. Sometimes sharing a semaphore by name is more appropriate than duplicating the handle.

Calls for up and down exist, although they have the somewhat odd names of ReleaseSemaphore (up) and WaitForSingleObject (down). It is also possible to give WaitForSingleObject a timeout, so the calling thread can be released eventually, even if the semaphore remains at 0 (although timers reintroduce races). WaitForSingleObject and WaitForMultipleObjects are the common interfaces used for waiting on the dispatcher objects discussed in Sec. 11.3. While it would have been possible to wrap the single-object version of these APIs in a wrapper with a somewhat more semaphore-friendly name, many threads use the multiple-object version which may include waiting for multiple flavors of synchronization objects as well as other events like process or thread termination, I/O completion, and messages being available on sockets and ports.

Mutexes are also kernel-mode objects used for synchronization, but simpler than semaphores because they do not have counters. They are essentially locks, with API functions for locking WaitForSingleObject and unlocking ReleaseMutex. Like semaphore handles, mutex handles can be duplicated and passed between processes so that threads in different processes can access the same mutex.

A third synchronization mechanism is called **critical sections**, which implement the concept of critical regions. These are similar to mutexes in Windows, except local to the address space of the creating thread. Because critical sections are not kernel-mode objects, they do not have explicit handles or security descriptors and cannot be passed between processes. Locking and unlocking are done with EnterCriticalSection and LeaveCriticalSection, respectively. Because these API functions are performed initially in user space and make kernel calls only when blocking is needed, they are much faster than mutexes. Critical sections are optimized to combine spin locks (on multiprocessors) with the use of kernel synchronization only when necessary. In many applications most critical sections are so rarely contended or have such short hold times that it is never necessary to allocate a kernel synchronization object. This results in a very significant savings in kernel memory.

Another synchronization mechanism we discuss uses kernel-mode objects called **events**. As we have described previously, there are two kinds: **notification events** and **synchronization events**. An event can be in one of two states: signaled or not-signaled. A thread can wait for an event to be signaled with WaitForSingleObject. If another thread signals an event with SetEvent, what happens depends on the type of event. With a notification event, all waiting threads are released and the event stays set until manually cleared with ResetEvent. With a synchronization event, if one or more threads are waiting, exactly one thread is released and

the event is cleared. An alternative operation is PulseEvent, which is like SetEvent except that if nobody is waiting, the pulse is lost and the event is cleared. In contrast, a SetEvent that occurs with no waiting threads is remembered by leaving the event in the signaled state so a subsequent thread that calls a wait API for the event will not actually wait.

The number of Win32 API calls dealing with processes, threads, and fibers is nearly 100, a substantial number of which deal with IPC in one form or another.

Two new synchronization primitives were recently added to Windows, WaitOnAddress and InitOnceExecuteOnce. WaitOnAddress is called to wait for the value at the specified address to be modified. The application must call either WakeByAddressSingle (or WakeByAddressAll) after modifying the location to wake either the first (or all) of the threads that called WaitOnAddress on that location. The advantage of this API over using events is that it is not necessary to allocate an explicit event for synchronization. Instead, the system hashes the address of the location to find a list of all the waiters for changes to a given address. WaitOnAddress functions similar to the sleep/wakeup mechanism found in the UNIX kernel. InitOnceExecuteOnce can be used to ensure that an initialization routine is run only once in a program. Correct initialization of data structures is surprisingly hard in multithreaded programs. A summary of the synchronization primitives discussed above, as well as some other important ones, is given in Fig. 11-24.

Note that not all of these are just system calls. While some are wrappers, others contain significant library code which maps the Win32 semantics onto the native NT APIs. Still others, like the fiber APIs, are purely user-mode functions since, as we mentioned earlier, kernel mode in Windows knows nothing about fibers. They are entirely implemented by user-mode libraries.

11.4.3 Implementation of Processes and Threads

In this section we will get into more detail about how Windows creates a process (and the initial thread). Because Win32 is the most documented interface, we will start there. But we will quickly work our way down into the kernel and understand the implementation of the native API call for creating a new process. We will focus on the main code paths that get executed whenever processes are created, as well as look at a few of the details that fill in gaps in what we have covered so far.

A process is created when another process makes the Win32 CreateProcess call. This call invokes a user-mode procedure in *kernel32.dll* that makes a call to NtCreateUserProcess in the kernel to create the process in several steps.

1. Convert the executable file name given as a parameter from a Win32 path name to an NT path name. If the executable has just a name without a directory path name, it is searched for in the directories listed in the default directories (which include, but are not limited to, those in the PATH variable in the environment).

Win32 API Function	Description
CreateProcess	Create a new process
CreateThread	Create a new thread in an existing process
CreateFiber	Create a new fiber
ExitProcess	Terminate current process and all its threads
ExitThread	Terminate this thread
ExitFiber	Terminate this fiber
SwitchToFiber	Run a different fiber on the current thread
SetPriorityClass	Set the priority class for a process
SetThreadPriority	Set the priority for one thread
CreateSemaphore	Create a new semaphore
CreateMutex	Create a new mutex
OpenSemaphore	Open an existing semaphore
OpenMutex	Open an existing mutex
WaitForSingleObject	Block on a single semaphore, mutex, etc.
WaitForMultipleObjects	Block on a set of objects whose handles are given
PulseEvent	Set an event to signaled, then to nonsignaled
ReleaseMutex	Release a mutex to allow another thread to acquire it
ReleaseSemaphore	Increase the semaphore count by 1
EnterCriticalSection	Acquire the lock on a critical section
LeaveCriticalSection	Release the lock on a critical section
WaitOnAddress	Block until the memory is changed at the specified address
WakeByAddressSingle	Wake the first thread that is waiting on this address
WakeByAddressAll	Wake all threads that are waiting on this address
InitOnceExecuteOnce	Ensure that an initialize routine executes only once

Figure 11-24. Some of the Win32 calls for managing processes, threads, and fibers.

2. Bundle up the process-creation parameters and pass them, along with the full path name of the executable program, to the native API NtCreateUserProcess.
3. Running in kernel mode, NtCreateUserProcess processes the parameters, then opens the program image and creates a section object that can be used to map the program into the new process' virtual address space.
4. The process manager allocates and initializes the process object (the kernel data structure representing a process to both the kernel and executive layers).

5. The memory manager creates the address space for the new process by allocating and initializing the page directories and the virtual address descriptors which describe the kernel-mode portion, including the process-specific regions, such as the **self-map** page-directory entries that gives each process kernel-mode access to the physical pages in its entire page table using kernel virtual addresses. (We will describe the self map in more detail in Sec. 11.5.)

6. A handle table is created for the new process, and all the handles from the caller that are allowed to be inherited are duplicated into it.

7. The shared user page is mapped, and the memory manager initializes the working-set data structures used for deciding what pages to trim from a process when physical memory is low. The pieces of the executable image represented by the section object are mapped into the new process' user-mode address space.

8. The executive creates and initializes the user-mode PEB, which is used by both user mode processes and the kernel to maintain processwide state information, such as the user-mode heap pointers and the list of loaded libraries (DLLs).

9. Virtual memory is allocated in the new process and used to pass parameters, including the environment strings and command line.

10. A process ID is allocated from the special handle table (ID table) the kernel maintains for efficiently allocating locally unique IDs for processes and threads.

11. A thread object is allocated and initialized. A user-mode stack is allocated along with the Thread Environment Block (TEB). The *CONTEXT* record which contains the thread's initial values for the CPU registers (including the instruction and stack pointers) is initialized.

12. The process object is added to the global list of processes. Handles for the process and thread objects are allocated in the caller's handle table. An ID for the initial thread is allocated from the ID table.

13. NtCreateUserProcess returns to user mode with the new process created, containing a single thread that is ready to run but suspended.

14. If the NT API fails, the Win32 code checks to see if this might be a process belonging to another subsystem like WOW64. Or perhaps the program is marked that it should be run under the debugger. These special cases are handled with special code in the user-mode CreateProcess code.

15. If NtCreateUserProcess was successful, there is still some work to be done. Win32 processes have to be registered with the Win32 subsystem process, *csrss.exe*. Kernel32.dll sends a message to *csrss* telling it about the new process along with the process and thread handles so it can duplicate itself. The process and threads are entered into the subsystems' tables so that they have a complete list of all Win32 processes and threads. The subsystem then displays a cursor containing a pointer with an hourglass to tell the user that something is going on but that the cursor can be used in the meanwhile. When the process makes its first GUI call, usually to create a window, the cursor is removed (it times out after 2 seconds if no call is forthcoming).
16. If the process is restricted, such as low-rights Internet Explorer, the token is modified to restrict what objects the new process can access.
17. If the application program was marked as needing to be shimmed to run compatibly with the current version of Windows, the specified *shims* are applied. **Shims** usually wrap library calls to slightly modify their behavior, such as returning a fake version number or delaying the freeing of memory.
18. Finally, call NtResumeThread to unsuspend the thread, and return the structure to the caller containing the IDs and handles for the process and thread that were just created.

In earlier versions of Windows, much of the algorithm for process creation was implemented in the user-mode procedure which would create a new process in using multiple system calls and by performing other work using the NT native APIs that support implementation of subsystems. These steps were moved into the kernel to reduce the ability of the parent process to manipulate the child process in the cases where the child is running a protected program, such as one that implements DRM to protect movies from piracy.

The original native API, NtCreateProcess, is still supported by the system, so much of process creation could still be done within user mode of the parent process—as long as the process being created is not a protected process.

Scheduling

The Windows kernel does not have a central scheduling thread. Instead, when a thread cannot run any more, the thread calls into the scheduler itself to see which thread to switch to. The following conditions invoke scheduling.

1. A running thread blocks on a semaphore, mutex, event, I/O, etc.
2. The thread signals an object (e.g., does an up on a semaphore).
3. The quantum expires.

In case 1, the thread is already in the kernel to carry out the operation on the dispatcher or I/O object. It cannot possibly continue, so it calls the scheduler code to pick its successor and load that thread's CONTEXT record to resume running it.

In case 2, the running thread is in the kernel, too. However, after signaling some object, it can definitely continue because signaling an object never blocks. Still, the thread is required to call the scheduler to see if the result of its action has released a thread with a higher scheduling priority that is now ready to run. If so, a thread switch occurs since Windows is fully preemptive (i.e., thread switches can occur at any moment, not just at the end of the current thread's quantum). However, in the case of a multicore chip or a multiprocessor, a thread that was made ready may be scheduled on a different CPU and the original thread can continue to execute on the current CPU even though its scheduling priority is lower.

In case 3, an interrupt to kernel mode occurs, at which point the thread executes the scheduler code to see who runs next. Depending on what other threads are waiting, the same thread may be selected, in which case it gets a new quantum and continues running. Otherwise a thread switch happens.

The scheduler is also called under two other conditions:

1. An I/O operation completes.
2. A timed wait expires.

In the first case, a thread may have been waiting on this I/O and is now released to run. A check has to be made to see if it should preempt the running thread since there is no guaranteed minimum run time. The scheduler is not run in the interrupt handler itself (since that may keep interrupts turned off too long). Instead, a DPC is queued for slightly later, after the interrupt handler is done. In the second case, a thread has done a down on a semaphore or blocked on some other object, but with a timeout that has now expired. Again it is necessary for the interrupt handler to queue a DPC to avoid having it run during the clock interrupt handler. If a thread has been made ready by this timeout, the scheduler will be run and if the newly runnable thread has higher priority, the current thread is preempted as in case 1.

Now we come to the actual scheduling algorithm. The Win32 API provides two APIs to influence thread scheduling. First, there is a call SetPriorityClass that sets the priority class of all the threads in the caller's process. The allowed values are: real-time, high, above normal, normal, below normal, and idle. The priority class determines the relative priorities of processes. The process priority class can also be used by a process to temporarily mark itself as being *background*, meaning that it should not interfere with any other activity in the system. Note that the priority class is established for the process, but it affects the actual priority of every thread in the process by setting a base priority that each thread starts with when created.

The second Win32 API is SetThreadPriority. It sets the relative priority of a thread (possibly, but not necessarily, the calling thread) with respect to the priority

class of its process. The allowed values are: time critical, highest, above normal, normal, below normal, lowest, and idle. Time-critical threads get the highest non-real-time scheduling priority, while idle threads get the lowest, irrespective of the priority class. The other priority values adjust the base priority of a thread with respect to the normal value determined by the priority class (+2, +1, 0, −1, −2, respectively). The use of priority classes and relative thread priorities makes it easier for applications to decide what priorities to specify.

The scheduler works as follows. The system has 32 priorities, numbered from 0 to 31. The combinations of priority class and relative priority are mapped onto 32 absolute thread priorities according to the table of Fig. 11-25. The number in the table determines the thread's **base priority**. In addition, every thread has a **current priority**, which may be higher (but not lower) than the base priority and which we will discuss shortly.

		Win32 process class priorities					
		Real-time	High	Above normal	Normal	Below normal	Idle
Win32 thread priorities	Time critical	31	15	15	15	15	15
	Highest	26	15	12	10	8	6
	Above normal	25	14	11	9	7	5
	Normal	24	13	10	8	6	4
	Below normal	23	12	9	7	5	3
	Lowest	22	11	8	6	4	2
	Idle	16	1	1	1	1	1

Figure 11-25. Mapping of Win32 priorities to Windows priorities.

To use these priorities for scheduling, the system maintains an array of 32 lists of threads, corresponding to priorities 0 through 31 derived from the table of Fig. 11-25. Each list contains ready threads at the corresponding priority. The basic scheduling algorithm consists of searching the array from priority 31 down to priority 0. As soon as a nonempty list is found, the thread at the head of the queue is selected and run for one quantum. If the quantum expires, the thread goes to the end of the queue at its priority level and the thread at the front is chosen next. In other words, when there are multiple threads ready at the highest priority level, they run round robin for one quantum each. If no thread is ready, the processor is idled—that is, set to a low power state waiting for an interrupt to occur.

It should be noted that scheduling is done by picking a thread without regard to which process that thread belongs. Thus, the scheduler does *not* first pick a process and then pick a thread in that process. It only looks at the threads. It does not consider which thread belongs to which process except to determine if it also needs to switch address spaces when switching threads.

To improve the scalability of the scheduling algorithm for multiprocessors with a high number of processors, the scheduler tries hard not to have to take the lock that protects access to the global array of priority lists. Instead, it sees if it can directly dispatch a thread that is ready to run to the processor where it should run.

For each thread the scheduler maintains the notion of its **ideal processor** and attempts to schedule it on that processor whenever possible. This improves the performance of the system, as the data used by a thread are more likely to already be available in the cache belonging to its ideal processor. The scheduler is aware of multiprocessors in which each CPU has its own memory and which can execute programs out of any memory—but at a cost if the memory is not local. These systems are called **NUMA** (**NonUniform Memory Access**) machines. The scheduler tries to optimize thread placement on such machines. The memory manager tries to allocate physical pages in the NUMA node belonging to the ideal processor for threads when they page fault.

The array of queue headers is shown in Fig. 11-26. The figure shows that there are actually four categories of priorities: real-time, user, zero, and idle, which is effectively –1. These deserve some comment. Priorities 16–31 are called system, and are intended to build systems that satisfy real-time constraints, such as deadlines needed for multimedia presentations. Threads with real-time priorities run before any of the threads with dynamic priorities, but not before DPCs and ISRs. If a real-time application wants to run on the system, it may require device drivers that are careful not to run DPCs or ISRs for any extended time as they might cause the real-time threads to miss their deadlines.

Ordinary users may not run real-time threads. If a user thread ran at a higher priority than, say, the keyboard or mouse thread and got into a loop, the keyboard or mouse thread would never run, effectively hanging the system. The right to set the priority class to real-time requires a special privilege to be enabled in the process' token. Normal users do not have this privilege.

Application threads normally run at priorities 1–15. By setting the process and thread priorities, an application can determine which threads get preference. The *ZeroPage* system threads run at priority 0 and convert free pages into pages of all zeroes. There is a separate *ZeroPage* thread for each real processor.

Each thread has a base priority based on the priority class of the process and the relative priority of the thread. But the priority used for determining which of the 32 lists a ready thread is queued on is determined by its current priority, which is normally the same as the base priority—but not always. Under certain conditions, the current priority of a nonreal-time thread is boosted by the kernel above the base priority (but never above priority 15). Since the array of Fig. 11-26 is based on the current priority, changing this priority affects scheduling. No adjustments are ever made to real-time threads.

Let us now see when a thread's priority is raised. First, when an I/O operation completes and releases a waiting thread, the priority is boosted to give it a chance to run again quickly and start more I/O. The idea here is to keep the I/O devices

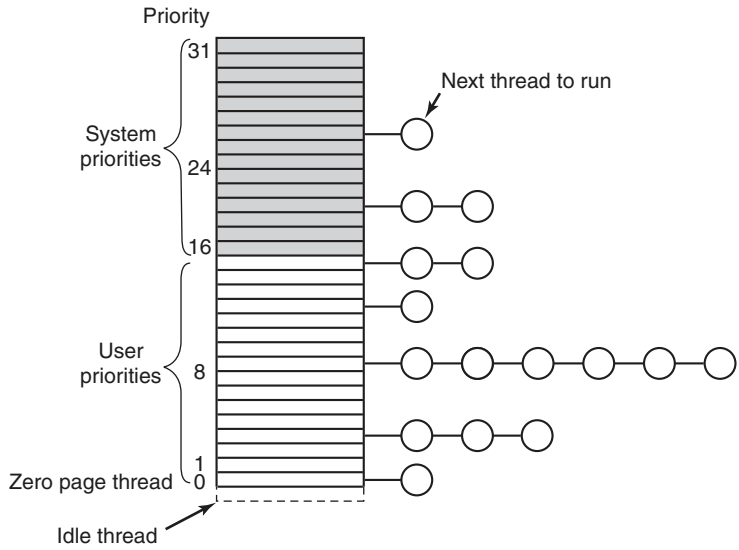

Figure 11-26. Windows supports 32 priorities for threads.

busy. The amount of boost depends on the I/O device, typically 1 for a disk, 2 for a serial line, 6 for the keyboard, and 8 for the sound card.

Second, if a thread was waiting on a semaphore, mutex, or other event, when it is released, it gets boosted by 2 levels if it is in the foreground process (the process controlling the window to which keyboard input is sent) and 1 level otherwise. This fix tends to raise interactive processes above the big crowd at level 8. Finally, if a GUI thread wakes up because window input is now available, it gets a boost for the same reason.

These boosts are not forever. They take effect immediately, and can cause rescheduling of the CPU. But if a thread uses all of its next quantum, it loses one priority level and moves down one queue in the priority array. If it uses up another full quantum, it moves down another level, and so on until it hits its base level, where it remains until it is boosted again.

There is one other case in which the system fiddles with the priorities. Imagine that two threads are working together on a producer-consumer type problem. The producer's work is harder, so it gets a high priority, say 12, compared to the consumer's 4. At a certain point, the producer has filled up a shared buffer and blocks on a semaphore, as illustrated in Fig. 11-27(a).

Before the consumer gets a chance to run again, an unrelated thread at priority 8 becomes ready and starts running, as shown in Fig. 11-27(b). As long as this thread wants to run, it will be able to, since it has a higher priority than the consumer, and the producer, though even higher, is blocked. Under these circumstances, the producer will never get to run again until the priority 8 thread gives up. This

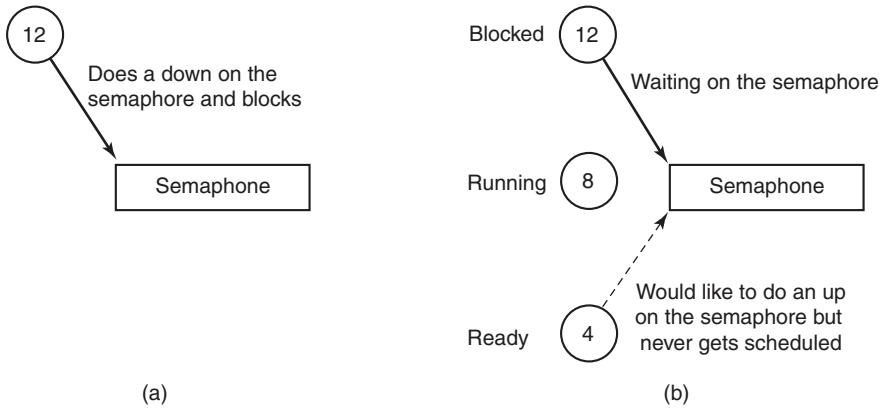

Figure 11-27. An example of priority inversion.

problem is well known under the name **priority inversion**. Windows addresses priority inversion between kernel threads through a facility in the thread scheduler called Autoboost. Autoboost automatically tracks resource dependencies between threads and boosts the scheduling priority of threads that hold resources needed by higher-priority threads.

Windows runs on PCs, which usually have only a single interactive session active at a time. However, Windows also supports a **terminal server** mode which supports multiple interactive sessions over the network using **RDP** (**Remote Desktop Protocol**). When running multiple user sessions, it is easy for one user to interfere with another by consuming too much processor resources. Windows implements a fair-share algorithm, **DFSS** (**Dynamic Fair-Share Scheduling**), which keeps sessions from running excessively. DFSS uses **scheduling groups** to organize the threads in each session. Within each group the threads are scheduled according to normal Windows scheduling policies, but each group is given more or less access to the processors based on how much the group has been running in aggregate. The relative priorities of the groups are adjusted slowly to allow ignore short bursts of activity and reduce the amount a group is allowed to run only if it uses excessive processor time over long periods.

11.5 MEMORY MANAGEMENT

Windows has an extremely sophisticated and complex virtual memory system. It has a number of Win32 functions for using it, implemented by the memory manager—the largest component of the NTOS executive layer. In the following sections we will look at the fundamental concepts, the Win32 API calls, and finally the implementation.

11.5.1 Fundamental Concepts

In Windows, every user process has its own virtual address space. For x86 machines, virtual addresses are 32 bits long, so each process has 4 GB of virtual address space, with the user and kernel each receiving 2 GB. For x64 machines, both the user and kernel receive more virtual addresses than they can reasonably use in the foreseeable future. For both x86 and x64, the virtual address space is demand paged, with a fixed page size of 4 KB—though in some cases, as we will see shortly, 2-MB large pages are also used (by using a page directory only and bypassing the corresponding page table).

The virtual address space layouts for three x86 processes are shown in Fig. 11-28 in simplified form. The bottom and top 64 KB of each process' virtual address space is normally unmapped. This choice was made intentionally to help catch programming errors and mitigate the exploitability of certain types of vulnerabilities.

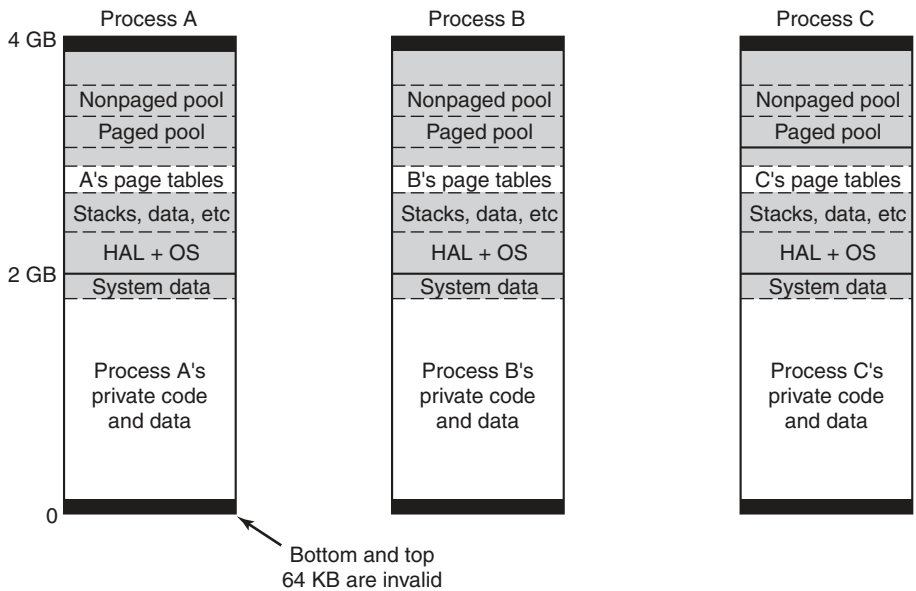

Figure 11-28. Virtual address space layout for three user processes on the x86. The white areas are private per process. The shaded areas are shared among all processes.

Starting at 64 KB comes the user's private code and data. This extends up to almost 2 GB. The upper 2 GB contains the operating system, including the code, data, and the paged and nonpaged pools. The upper 2 GB is the kernel's virtual memory and is shared among all user processes, except for virtual memory data like the page tables and working-set lists, which are per-process. Kernel virtual

memory is accessible only while running in kernel mode. The reason for sharing the process' virtual memory with the kernel is that when a thread makes a system call, it traps into kernel mode and can continue running without changing the memory map. All that has to be done is switch to the thread's kernel stack. From a performance point of view, this is a big win, and something UNIX does as well. Because the process' user-mode pages are still accessible, the kernel-mode code can read parameters and access buffers without having to switch back and forth between address spaces or temporarily double-map pages into both. The trade-off here is less private address space per process in return for faster system calls.

Windows allows threads to attach themselves to other address spaces while running in the kernel. Attachment to an address space allows the thread to access all of the user-mode address space, as well as the portions of the kernel address space that are specific to a process, such as the self-map for the page tables. Threads must switch back to their original address space before returning to user mode.

Virtual Address Allocation

Each page of virtual addresses can be in one of three states: invalid, reserved, or committed. An **invalid page** is not currently mapped to a memory section object and a reference to it causes a page fault that results in an access violation. Once code or data is mapped onto a virtual page, the page is said to be **committed**. A page fault on a committed page results in mapping the page containing the virtual address that caused the fault onto one of the pages represented by the section object or stored in the pagefile. Often this will require allocating a physical page and performing I/O on the file represented by the section object to read in the data from disk. But page faults can also occur simply because the page-table entry needs to be updated, as the physical page referenced is still cached in memory, in which case I/O is not required. These are called **soft faults** and we will discuss them in more detail shortly.

A virtual page can also be in the **reserved** state. A reserved virtual page is invalid but has the property that those virtual addresses will never be allocated by the memory manager for another purpose. As an example, when a new thread is created, many pages of user-mode stack space are reserved in the process' virtual address space, but only one page is committed. As the stack grows, the virtual memory manager will automatically commit additional pages under the covers, until the reservation is almost exhausted. The reserved pages function as guard pages to keep the stack from growing too far and overwriting other process data. Reserving all the virtual pages means that the stack can eventually grow to its maximum size without the risk that some of the contiguous pages of virtual address space needed for the stack might be given away for another purpose. In addition to the invalid, reserved, and committed attributes, pages also have other attributes, such as being readable, writable, and executable.

Pagefiles

An interesting trade-off occurs with assignment of backing store to committed pages that are not being mapped to specific files. These pages use the **pagefile**. The question is *how* and *when* to map the virtual page to a specific location in the pagefile. A simple strategy would be to assign each virtual page to a page in one of the paging files on disk at the time the virtual page was committed. This would guarantee that there was always a known place to write out each committed page should it be necessary to evict it from memory.

Windows uses a *just-in-time* strategy. Committed pages that are backed by the pagefile are not assigned space in the pagefile until the time that they have to be paged out. No disk space is allocated for pages that are never paged out. If the total virtual memory is less than the available physical memory, a pagefile is not needed at all. This is convenient for embedded systems based on Windows. It is also the way the system is booted, since pagefiles are not initialized until the first user-mode process, *smss.exe*, begins running.

With a preallocation strategy the total virtual memory in the system used for private data (stacks, heap, and copy-on-write code pages) is limited to the size of the pagefiles. With just-in-time allocation the total virtual memory can be almost as large as the combined size of the pagefiles and physical memory. With disks so large and cheap vs. physical memory, the savings in space is not as significant as the increased performance that is possible.

With demand-paging, requests to read pages from disk need to be initiated right away, as the thread that encountered the missing page cannot continue until this *page-in* operation completes. The possible optimizations for faulting pages into memory involve attempting to prepage additional pages in the same I/O operation. However, operations that write modified pages to disk are not normally synchronous with the execution of threads. The just-in-time strategy for allocating pagefile space takes advantage of this to boost the performance of writing modified pages to the pagefile. Modified pages are grouped together and written in big chunks. Since the allocation of space in the pagefile does not happen until the pages are being written, the number of seeks required to write a batch of pages can be optimized by allocating the pagefile pages to be near each other, or even making them contiguous.

When pages stored in the pagefile are read into memory, they keep their allocation in the pagefile until the first time they are modified. If a page is never modified, it will go onto a special list of free physical pages, called the **standby list**, where it can be reused without having to be written back to disk. If it *is* modified, the memory manager will free the pagefile page and the only copy of the page will be in memory. The memory manager implements this by marking the page as read-only after it is loaded. The first time a thread attempts to write the page the memory manager will detect this situation and free the pagefile page, grant write access to the page, and then have the thread try again.

Windows supports up to 16 pagefiles, normally spread out over separate disks to achieve higher I/O bandwidth. Each one has an initial size and a maximum size it can grow to later if needed, but it is better to create these files to be the maximum size at system installation time. If it becomes necessary to grow a pagefile when the file system is much fuller, it is likely that the new space in the pagefile will be highly fragmented, reducing performance.

The operating system keeps track of which virtual page maps onto which part of which paging file by writing this information into the page-table entries for the process for private pages, or into prototype page-table entries associated with the section object for shared pages. In addition to the pages that are backed by the pagefile, many pages in a process are mapped to regular files in the file system.

The executable code and read-only data in a program file (e.g., an EXE or DLL) can be mapped into the address space of whatever process is using it. Since these pages cannot be modified, they never need to be paged out but the physical pages can just be immediately reused after the page-table mappings are all marked as invalid. When the page is needed again in the future, the memory manager will read the page in from the program file.

Sometimes pages that start out as read-only end up being modified, for example, setting a breakpoint in the code when debugging a process, or fixing up code to relocate it to different addresses within a process, or making modifications to data pages that started out shared. In cases like these, Windows, like most modern operating systems, supports a type of page called **copy-on-write**. These pages start out as ordinary mapped pages, but when an attempt is made to modify any part of the page the memory manager makes a private, writable copy. It then updates the page table for the virtual page so that it points at the private copy and has the thread retry the write—which will now succeed. If that copy later needs to be paged out, it will be written to the pagefile rather than the original file,

Besides mapping program code and data from EXE and DLL files, ordinary files can be mapped into memory, allowing programs to reference data from files without doing read and write operations. I/O operations are still needed, but they are provided implicitly by the memory manager using the section object to represent the mapping between pages in memory and the blocks in the files on disk.

Section objects do not have to refer to a file. They can refer to anonymous regions of memory. By mapping anonymous section objects into multiple processes, memory can be shared without having to allocate a file on disk. Since sections can be given names in the NT namespace, processes can rendezvous by opening sections by name, as well as by duplicating and passing handles between processes.

11.5.2 Memory-Management System Calls

The Win32 API contains a number of functions that allow a process to manage its virtual memory explicitly. The most important of these functions are listed in Fig. 11-29. All of them operate on a region consisting of either a single page or a

sequence of two or more pages that are consecutive in the virtual address space. Of course, processes do not have to manage their memory; paging happens automatically, but these calls give processes additional power and flexibility.

Win32 API function	Description
VirtualAlloc	Reserve or commit a region
VirtualFree	Release or decommit a region
VirtualProtect	Change the read/write/execute protection on a region
VirtualQuery	Inquire about the status of a region
VirtualLock	Make a region memory resident (i.e., disable paging for it)
VirtualUnlock	Make a region pageable in the usual way
CreateFileMapping	Create a file-mapping object and (optionally) assign it a name
MapViewOfFile	Map (part of) a file into the address space
UnmapViewOfFile	Remove a mapped file from the address space
OpenFileMapping	Open a previously created file-mapping object

Figure 11-29. The principal Win32 API functions for managing virtual memory in Windows.

The first four API functions are used to allocate, free, protect, and query regions of virtual address space. Allocated regions always begin on 64-KB boundaries to minimize porting problems to future architectures with pages larger than current ones. The actual amount of address space allocated can be less than 64 KB, but must be a multiple of the page size. The next two APIs give a process the ability to hardwire pages in memory so they will not be paged out and to undo this property. A real-time program might need pages with this property to avoid page faults to disk during critical operations, for example. A limit is enforced by the operating system to prevent processes from getting too greedy. The pages actually can be removed from memory, but only if the entire process is swapped out. When it is brought back, all the locked pages are reloaded before any thread can start running again. Although not shown in Fig. 11-29, Windows also has native API functions to allow a process to access the virtual memory of a different process over which it has been given control, that is, for which it has a handle (see Fig. 11-7).

The last four API functions listed are for managing memory-mapped files. To map a file, a file-mapping object must first be created with CreateFileMapping (see Fig. 11-8). This function returns a handle to the file-mapping object (i.e., a section object) and optionally enters a name for it into the Win32 namespace so that other processes can use it, too. The next two functions map and unmap views on section objects from a process' virtual address space. The last API can be used by a process to map share a mapping that another process created with CreateFileMapping, usually one created to map anonymous memory. In this way, two or more processes can share regions of their address spaces. This technique allows them to write in limited regions of each other's virtual memory.

11.5.3 Implementation of Memory Management

Windows, on the x86, supports a single linear 4-GB demand-paged address space per process. Segmentation is not supported in any form. Theoretically, page sizes can be any power of 2 up to 64 KB. On the x86 they are normally fixed at 4 KB. In addition, the operating system can use 2-MB large pages to improve the effectiveness of the **TLB** (**Translation Lookaside Buffer**) in the processor's memory management unit. Use of 2-MB large pages by the kernel and large applications significantly improves performance by improving the hit rate for the TLB and reducing the number of times the page tables have to be walked to find entries that are missing from the TLB.

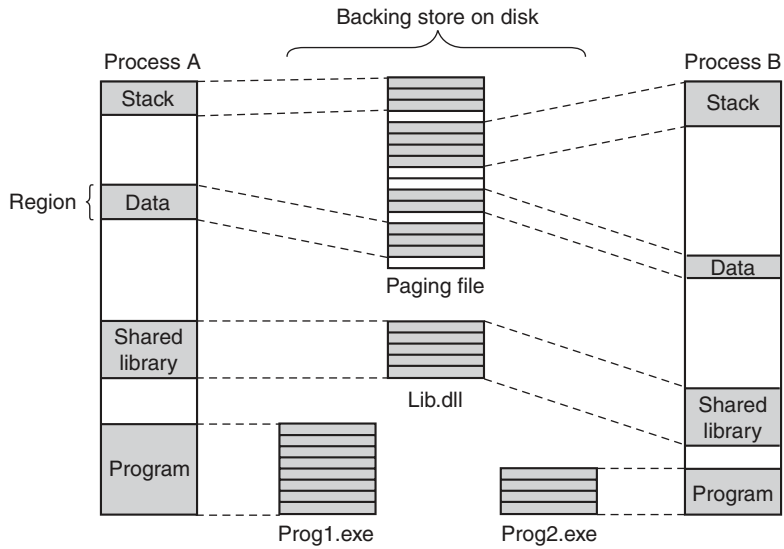

Figure 11-30. Mapped regions with their shadow pages on disk. The *lib.dll* file is mapped into two address spaces at the same time.

Unlike the scheduler, which selects individual threads to run and does not care much about processes, the memory manager deals entirely with processes and does not care much about threads. After all, processes, not threads, own the address space and that is what the memory manager is concerned with. When a region of virtual address space is allocated, as four of them have been for process *A* in Fig. 11-30, the memory manager creates a **VAD** (**Virtual Address Descriptor**) for it, listing the range of addresses mapped, the section representing the backing store file and offset where it is mapped, and the permissions. When the first page is touched, the directory of page tables is created and its physical address is inserted into the process object. An address space is completely defined by the list of its VADs. The VADs are organized into a balanced tree, so that the descriptor for a

particular address can be found efficiently. This scheme supports sparse address spaces. Unused areas between the mapped regions use no resources (memory or disk) so they are essential free.

Page-Fault Handling

When a process starts on Windows, many of the pages mapping the program's EXE and DLL image files may already be in memory because they are shared with other processes. The writable pages of the images are marked *copy-on-write* so that they can be shared up to the point they need to be modified. If the operating system recognizes the EXE from a previous execution, it may have recorded the page-reference pattern, using a technology Microsoft calls **SuperFetch**. SuperFetch attempts to prepage many of the needed pages even though the process has not faulted on them yet. This reduces the latency for starting up applications by overlapping the reading of the pages from disk with the execution of the initialization code in the images. It improves throughput to disk because it is easier for the disk drivers to organize the reads to reduce the seek time needed. Process prepaging is also used during boot of the system, when a background application moves to the foreground, and when restarting the system after hibernation.

Prepaging is supported by the memory manager, but implemented as a separate component of the system. The pages brought in are not inserted into the process' page table, but instead are inserted into the *standby list* from which they can quickly be inserted into the process as needed without accessing the disk.

Nonmapped pages are slightly different in that they are not initialized by reading from the file. Instead, the first time a nonmapped page is accessed the memory manager provides a new physical page, making sure the contents are all zeroes (for security reasons). On subsequent faults a nonmapped page may need to be found in memory or else must be read back from the pagefile.

Demand paging in the memory manager is driven by page faults. On each page fault, a trap to the kernel occurs. The kernel then builds a machine-independent descriptor telling what happened and passes this to the memory-manager part of the executive. The memory manager then checks the access for validity. If the faulted page falls within a committed region, it looks up the address in the list of VADs and finds (or creates) the process page-table entry. In the case of a shared page, the memory manager uses the prototype page-table entry associated with the section object to fill in the new page-table entry for the process page table.

The format of the page-table entries differs depending on the processor architecture. For the x86 and x64, the entries for a mapped page are shown in Fig. 11-31. If an entry is marked valid, its contents are interpreted by the hardware so that the virtual address can be translated into the correct physical page. Unmapped pages also have entries, but they are marked *invalid* and the hardware ignores the rest of the entry. The software format is somewhat different from the hardware

format and is determined by the memory manager. For example, for an unmapped page that must be allocated and zeroed before it may be used, that fact is noted in the page-table entry.

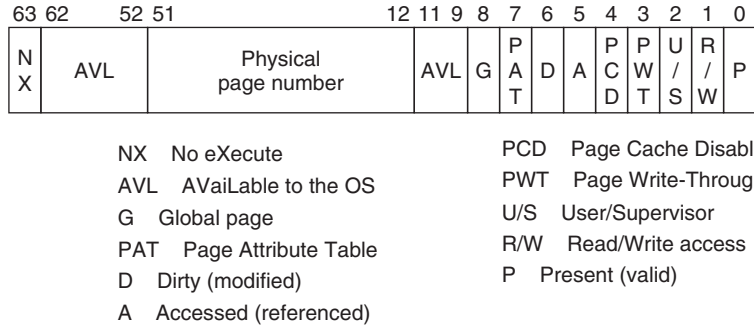

Figure 11-31. A page-table entry (PTE) for a mapped page on the Intel x86 and AMD x64 architectures.

Two important bits in the page-table entry are updated by the hardware directly. These are the access (A) and dirty (D) bits. These bits keep track of when a particular page mapping has been used to access the page and whether that access could have modified the page by writing it. This really helps the performance of the system because the memory manager can use the access bit to implement the **LRU** (**Least-Recently Used**) style of paging. The LRU principle says that pages which have not been used the longest are the least likely to be used again soon. The access bit allows the memory manager to determine that a page has been accessed. The dirty bit lets the memory manager know that a page may have been modified, or more significantly, that a page has *not* been modified. If a page has not been modified since being read from disk, the memory manager does not have to write the contents of the page to disk before using it for something else.

Both the x86 and x64 use a 64-bit page-table entry, as shown in Fig. 11-31.

Each page fault can be considered as being in one of five categories:

1. The page referenced is not committed.
2. Access to a page has been attempted in violation of the permissions.
3. A shared copy-on-write page was about to be modified.
4. The stack needs to grow.
5. The page referenced is committed but not currently mapped in.

The first and second cases are due to programming errors. If a program attempts to use an address which is not supposed to have a valid mapping, or attempts an invalid operation (like attempting to write a read-only page) this is called

an **access violation** and usually results in termination of the process. Access violations are often the result of bad pointers, including accessing memory that was freed and unmapped from the process.

The third case has the same symptoms as the second one (an attempt to write to a read-only page), but the treatment is different. Because the page has been marked as *copy-on-write*, the memory manager does not report an access violation, but instead makes a private copy of the page for the current process and then returns control to the thread that attempted to write the page. The thread will retry the write, which will now complete without causing a fault.

The fourth case occurs when a thread pushes a value onto its stack and crosses onto a page which has not been allocated yet. The memory manager is programmed to recognize this as a special case. As long as there is still room in the virtual pages reserved for the stack, the memory manager will supply a new physical page, zero it, and map it into the process. When the thread resumes running, it will retry the access and succeed this time around.

Finally, the fifth case is a normal page fault. However, it has several subcases. If the page is mapped by a file, the memory manager must search its data structures, such as the prototype page table associated with the section object to be sure that there is not already a copy in memory. If there is, say in another process or on the standby or modified page lists, it will just share it—perhaps marking it as copy-on-write if changes are not supposed to be shared. If there is not already a copy, the memory manager will allocate a free physical page and arrange for the file page to be copied in from disk, unless another the page is already transitioning in from disk, in which case it is only necessary to wait for the transition to complete.

When the memory manager can satisfy a page fault by finding the needed page in memory rather than reading it in from disk, the fault is classified as a **soft fault**. If the copy from disk is needed, it is a **hard fault**. Soft faults are much cheaper, and have little impact on application performance compared to hard faults. Soft faults can occur because a shared page has already been mapped into another process, or only a new zero page is needed, or the needed page was trimmed from the process' working set but is being requested again before it has had a chance to be reused. Soft faults can also occur because pages have been compressed to effectively increase the size of physical memory. For most configurations of CPU, memory, and I/O in current systems it is more efficient to use compression rather than incur the I/O expense (performance and energy) required to read a page from disk.

When a physical page is no longer mapped by the page table in any process it goes onto one of three lists: free, modified, or standby. Pages that will never be needed again, such as stack pages of a terminating process, are freed immediately. Pages that may be faulted again go to either the modified list or the standby list, depending on whether or not the dirty bit was set for any of the page-table entries that mapped the page since it was last read from disk. Pages in the modified list will be eventually written to disk, then moved to the standby list.

The memory manager can allocate pages as needed using either the free list or the standby list. Before allocating a page and copying it in from disk, the memory manager always checks the standby and modified lists to see if it already has the page in memory. The prepaging scheme in Windows thus converts future hard faults into soft faults by reading in the pages that are expected to be needed and pushing them onto the standby list. The memory manager itself does a small amount of ordinary prepaging by accessing groups of consecutive pages rather than single pages. The additional pages are immediately put on the standby list. This is not generally wasteful because the overhead in the memory manager is very much dominated by the cost of doing a single I/O. Reading a cluster of pages rather than a single page is negligibly more expensive.

The page-table entries in Fig. 11-31 refer to physical page numbers, not virtual page numbers. To update page-table (and page-directory) entries, the kernel needs to use virtual addresses. Windows maps the page tables and page directories for the current process into kernel virtual address space using self-map entries in the page directory, as shown in Fig. 11-32. By making page-directory entries point at the page directory (the self-map), there are virtual addresses that can be used to refer to page-directory entries (a) as well as page table entries (b). The self-map occupies the same 8 MB of kernel virtual addresses for every process (on the x86). For simplicity the figure shows the x86 self-map for 32-bit **PTEs** (**Page-Table Entries**). Windows actually uses 64-bit PTEs so the system can makes use of more than 4 GB of physical memory. With 32-bit PTEs, the self-map uses only one **PDE** (**Page-Directory Entry**) in the page directory, and thus occupies only 4 MB of addresses rather than 8 MB.

The Page Replacement Algorithm

When the number of free physical memory pages starts to get low, the memory manager starts working to make more physical pages available by removing them from user-mode processes as well as the system process, which represents kernel-mode use of pages. The goal is to have the most important virtual pages present in memory and the others on disk. The trick is in determining what *important* means. In Windows this is answered by making heavy use of the working-set concept. Each process (*not* each thread) has a working set. This set consists of the mapped-in pages that are in memory and thus can be referenced without a page fault. The size and composition of the working set fluctuates as the process' threads run, of course.

Each process' working set is described by two parameters: the minimum size and the maximum size. These are not hard bounds, so a process may have fewer pages in memory than its minimum or (under certain circumstances) more than its maximum. Every process starts with the same minimum and maximum, but these bounds can change over time, or can be determined by the job object for processes contained in a job. The default initial minimum is in the range 20–50 pages and

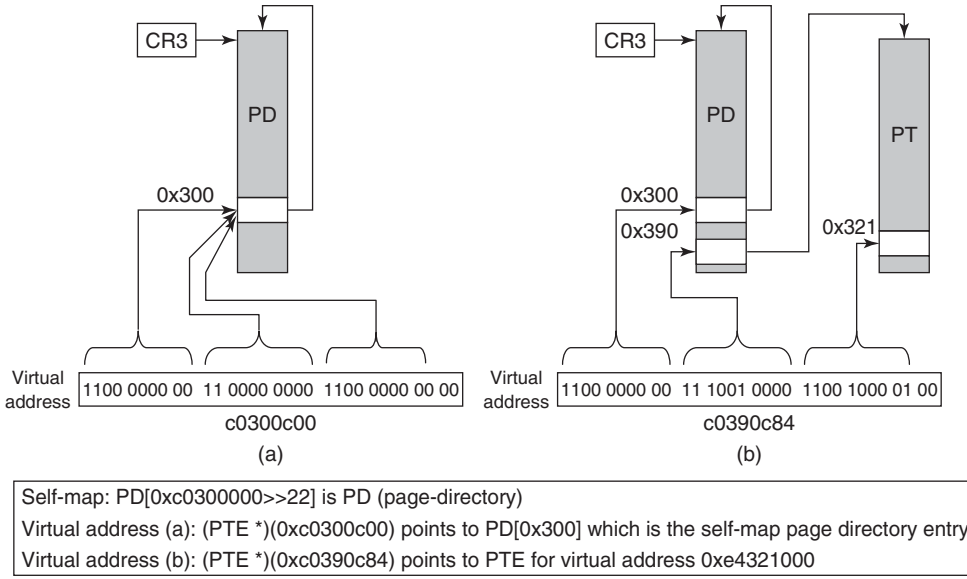

Figure 11-32. The Windows self-map entries are used to map the physical pages of the page tables and page directory into kernel virtual addresses (shown for 32-bit PTEs).

the default initial maximum is in the range 45–345 pages, depending on the total amount of physical memory in the system. The system administrator can change these defaults, however. While few home users will try, server admins might.

Working sets come into play only when the available physical memory is getting low in the system. Otherwise processes are allowed to consume memory as they choose, often far exceeding the working-set maximum. But when the system comes under **memory pressure**, the memory manager starts to squeeze processes back into their working sets, starting with processes that are over their maximum by the most. There are three levels of activity by the working-set manager, all of which is periodic based on a timer. New activity is added at each level:

1. **Lots of memory available:** Scan pages resetting access bits and using their values to represent the *age* of each page. Keep an estimate of the unused pages in each working set.
2. **Memory getting tight:** For any process with a significant proportion of unused pages, stop adding pages to the working set and start replacing the oldest pages whenever a new page is needed. The replaced pages go to the standby or modified list.
3. **Memory is tight:** Trim (i.e., reduce) working sets to be below their maximum by removing the oldest pages.

The working set manager runs every second, called from the **balance set manager** thread. The working-set manager throttles the amount of work it does to keep from overloading the system. It also monitors the writing of pages on the modified list to disk to be sure that the list does not grow too large, waking the ModifiedPageWriter thread as needed.

Physical Memory Management

Above we mentioned three different lists of physical pages, the free list, the standby list, and the modified list. There is a fourth list which contains free pages that have been zeroed. The system frequently needs pages that contain all zeros. When new pages are given to processes, or the final partial page at the end of a file is read, a zero page is needed. It is time consuming to write a page with zeros, so it is better to create zero pages in the background using a low-priority thread. There is also a fifth list used to hold pages that have been detected as having hardware errors (i.e., through hardware error detection).

All pages in the system either are referenced by a valid page-table entry or are on one of these five lists, which are collectively called the **PFN database** (**Page Frame Number database**). Fig. 11-33 shows the structure of the PFN Database. The table is indexed by physical page-frame number. The entries are fixed length, but different formats are used for different kinds of entries (e.g., shared vs. private). Valid entries maintain the page's state and a count of how many page tables point to the page, so that the system can tell when the page is no longer in use. Pages that are in a working set tell which entry references them. There is also a pointer to the process page table that points to the page (for nonshared pages) or to the prototype page table (for shared pages).

Additionally there is a link to the next page on the list (if any), and various other fields and flags, such as *read in progress*, *write in progress*, and so on. To save space, the lists are linked together with fields referring to the next element by its index within the table rather than pointers. The table entries for the physical pages are also used to summarize the dirty bits found in the various page table entries that point to the physical page (i.e., because of shared pages). There is also information used to represent differences in memory pages on larger server systems which have memory that is faster from some processors than from others, namely NUMA machines.

Pages are moved between the working sets and the various lists by the working-set manager and other system threads. Let us examine the transitions. When the working-set manager removes a page from a working set, the page goes on the bottom of the standby or modified list, depending on its state of cleanliness. This transition is shown as (1) in Fig. 11-34.

Pages on both lists are still valid pages, so if a page fault occurs and one of these pages is needed, it is removed from the list and faulted back into the working set without any disk I/O (2). When a process exits, its nonshared pages cannot be

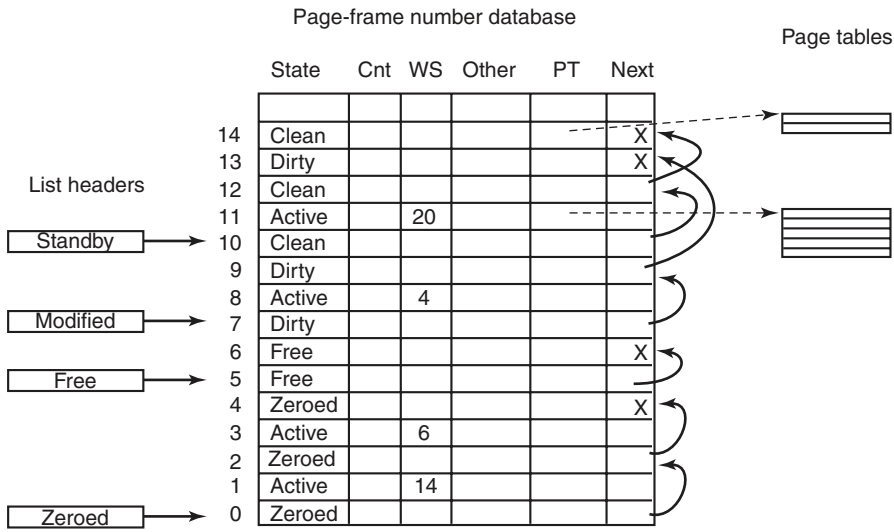

Figure 11-33. Some of the major fields in the page-frame database for a valid page.

faulted back to it, so the valid pages in its page table and any of its pages on the modified or standby lists go on the free list (3). Any pagefile space in use by the process is also freed.

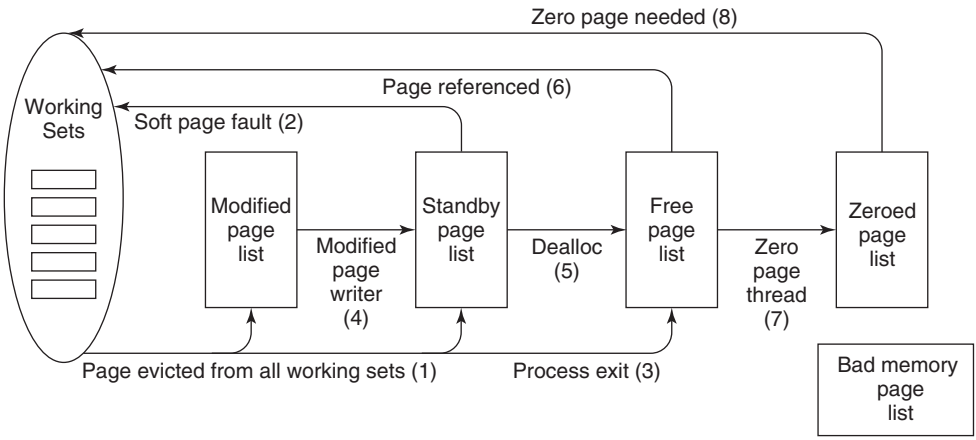

Figure 11-34. The various page lists and the transitions between them.

Other transitions are caused by other system threads. Every 4 seconds the balance set manager thread runs and looks for processes all of whose threads have been idle for a certain number of seconds. If it finds any such processes, their

kernel stacks are unpinned from physical memory and their pages are moved to the standby or modified lists, also shown as (1).

Two other system threads, the **mapped page writer** and the **modified page writer**, wake up periodically to see if there are enough clean pages. If not, they take pages from the top of the modified list, write them back to disk, and then move them to the standby list (4). The former handles writes to mapped files and the latter handles writes to the pagefiles. The result of these writes is to transform modified (dirty) pages into standby (clean) pages.

The reason for having two threads is that a mapped file might have to grow as a result of the write, and growing it requires access to on-disk data structures to allocate a free disk block. If there is no room in memory to bring them in when a page has to be written, a deadlock could result. The other thread can solve the problem by writing out pages to a paging file.

The other transitions in Fig. 11-34 are as follows. If a process unmaps a page, the page is no longer associated with a process and can go on the free list (5), except for the case that it is shared. When a page fault requires a page frame to hold the page about to be read in, the page frame is taken from the free list (6), if possible. It does not matter that the page may still contain confidential information because it is about to be overwritten in its entirety.

The situation is different when a stack grows. In that case, an empty page frame is needed and the security rules require the page to contain all zeros. For this reason, another kernel system thread, the **ZeroPage thread**, runs at the lowest priority (see Fig. 11-26), erasing pages that are on the free list and putting them on the zeroed page list (7). Whenever the CPU is idle and there are free pages, they might as well be zeroed since a zeroed page is potentially more useful than a free page and it costs nothing to zero the page when the CPU is idle.

The existence of all these lists leads to some subtle policy choices. For example, suppose that a page has to be brought in from disk and the free list is empty. The system is now forced to choose between taking a clean page from the standby list (which might otherwise have been faulted back in later) or an empty page from the zeroed page list (throwing away the work done in zeroing it). Which is better?

The memory manager has to decide how aggressively the system threads should move pages from the modified list to the standby list. Having clean pages around is better than having dirty pages around (since clean ones can be reused instantly), but an aggressive cleaning policy means more disk I/O and there is some chance that a newly cleaned page may be faulted back into a working set and dirtied again anyway. In general, Windows resolves these kinds of trade-offs through algorithms, heuristics, guesswork, historical precedent, rules of thumb, and administrator-controlled parameter settings.

Modern Windows introduced an additional abstraction layer at the bottom of the memory manager, called the **store manager**. This layer makes decisions about how to optimize the I/O operations to the available backing stores. Persistent storage systems include auxiliary flash memory and SSDs in addition to rotating disks.

The store manager optimizes where and how physical memory pages are backed by the persistent stores in the system. It also implements optimization techniques such as copy-on-write sharing of identical physical pages and compression of the pages in the standby list to effectively increase the available RAM.

Another change in memory management in Modern Windows is the introduction of a **swap file**. Historically memory management in Windows has been based on working sets, as described above. As memory pressure increases, the memory manager squeezes on the working sets to reduce the footprint each process has in memory. The modern application model introduces opportunities for new efficiencies. Since the process containing the foreground part of a modern application is no longer given processor resources once the user has switched away, there is no need for its pages to be resident. As memory pressure builds in the system, the pages in the process may be removed as part of normal working-set management. However, the process lifetime manager knows how long it has been since the user switched to the application's foreground process. When more memory is needed it picks a process that has not run in a while and calls into the memory manager to efficiently swap all the pages in a small number of I/O operations. The pages will be written to the swap file by aggregating them into one or more large chunks. This means that the entire process can also be restored in memory with fewer I/O operations.

All in all, memory management is a highly complex executive component with many data structures, algorithms, and heuristics. It attempts to be largely self tuning, but there are also many knobs that administrators can tweak to affect system performance. A number of these knobs and the associated counters can be viewed using tools in the various tool kits mentioned earlier. Probably the most important thing to remember here is that memory management in real systems is a lot more than just one simple paging algorithm like clock or aging.

11.6 CACHING IN WINDOWS

The Windows cache improves the performance of file systems by keeping recently and frequently used regions of files in memory. Rather than cache physical addressed blocks from the disk, the cache manager manages virtually addressed blocks, that is, regions of files. This approach fits well with the structure of the native NT File System (NTFS), as we will see in Sec. 11.8. NTFS stores all of its data as files, including the file-system metadata.

The cached regions of files are called *views* because they represent regions of kernel virtual addresses that are mapped onto file-system files. Thus, the actual management of the physical memory in the cache is provided by the memory manager. The role of the cache manager is to manage the use of kernel virtual addresses for views, arrange with the memory manager to *pin* pages in physical memory, and provide interfaces for the file systems.

The Windows cache-manager facilities are shared among all the file systems. Because the cache is virtually addressed according to individual files, the cache manager is easily able to perform read-ahead on a per-file basis. Requests to access cached data come from each file system. Virtual caching is convenient because the file systems do not have to first translate file offsets into physical block numbers before requesting a cached file page. Instead, the translation happens later when the memory manager calls the file system to access the page on disk.

Besides management of the kernel virtual address and physical memory resources used for caching, the cache manager also has to coordinate with file systems regarding issues like coherency of views, flushing to disk, and correct maintenance of the end-of-file marks—particularly as files expand. One of the most difficult aspects of a file to manage between the file system, the cache manager, and the memory manager is the offset of the last byte in the file, called the *ValidDataLength*. If a program writes past the end of the file, the blocks that were skipped have to be filled with zeros, and for security reasons it is critical that the ValidDataLength recorded in the file metadata not allow access to uninitialized blocks, so the zero blocks have to be written to disk before the metadata is updated with the new length. While it is expected that if the system crashes, some of the blocks in the file might not have been updated from memory, it is not acceptable that some of the blocks might contain data previously belonging to other files.

Let us now examine how the cache manager works. When a file is referenced, the cache manager maps a 256-KB chunk of kernel virtual address space onto the file. If the file is larger than 256 KB, only a portion of the file is mapped at a time. If the cache manager runs out of 256-KB chunks of virtual address space, it must unmap an old file before mapping in a new one. Once a file is mapped, the cache manager can satisfy requests for its blocks by just copying from kernel virtual address space to the user buffer. If the block to be copied is not in physical memory, a page fault will occur and the memory manager will satisfy the fault in the usual way. The cache manager is not even aware of whether the block was in memory or not. The copy always succeeds.

The cache manager also works for pages that are mapped into virtual memory and accessed with pointers rather than being copied between kernel and user-mode buffers. When a thread accesses a virtual address mapped to a file and a page fault occurs, the memory manager may in many cases be able to satisfy the access as a soft fault. It does not need to access the disk, since it finds that the page is already in physical memory because it is mapped by the cache manager.

11.7 INPUT/OUTPUT IN WINDOWS

The goals of the Windows I/O manager are to provide a fundamentally extensive and flexible framework for efficiently handling a very wide variety of I/O devices and services, support automatic device discovery and driver installation (plug

and play) and power management for devices and the CPU—all using a fundamentally asynchronous structure that allows computation to overlap with I/O transfers. There are many hundreds of thousands of devices that work with Windows. For a large number of common devices it is not even necessary to install a driver, because there is already a driver that shipped with the Windows operating system. But even so, counting all the revisions, there are almost a million distinct driver binaries that run on Windows. In the following sections we will examine some of the issues relating to I/O.

11.7.1 Fundamental Concepts

The I/O manager is on intimate terms with the plug-and-play manager. The basic idea behind plug and play is that of an enumerable bus. Many buses, including PC Card, PCI, PCIe, AGP, USB, IEEE 1394, EIDE, SCSI, and SATA, have been designed so that the plug-and-play manager can send a request to each slot and ask the device there to identify itself. Having discovered what is out there, the plug-and-play manager allocates hardware resources, such as interrupt levels, locates the appropriate drivers, and loads them into memory. As each driver is loaded, a driver object is created for it. And then for each device, at least one device object is allocated. For some buses, such as SCSI, enumeration happens only at boot time, but for other buses, such as USB, it can happen at any time, requiring close cooperation between the plug-and-play manager, the bus drivers (which actually do the enumerating), and the I/O manager.

In Windows, all the file systems, antivirus filters, volume managers, network protocol stacks, and even kernel services that have no associated hardware are implemented using I/O drivers. The system configuration must be set to cause some of these drivers to load, because there is no associated device to enumerate on the bus. Others, like the file systems, are loaded by special code that detects they are needed, such as the file-system recognizer that looks at a raw volume and deciphers what type of file system format it contains.

An interesting feature of Windows is its support for **dynamic disks**. These disks may span multiple partitions and even multiple disks and may be reconfigured on the fly, without even having to reboot. In this way, logical volumes are no longer constrained to a single partition or even a single disk so that a single file system may span multiple drives in a transparent way.

The I/O to volumes can be filtered by a special Windows driver to produce **Volume Shadow Copies**. The filter driver creates a snapshot of the volume which can be separately mounted and represents a volume at a previous point in time. It does this by keeping track of changes after the snapshot point. This is very convenient for recovering files that were accidentally deleted, or traveling back in time to see the state of a file at periodic snapshots made in the past.

But shadow copies are also valuable for making accurate backups of server systems. The operating system works with server applications to have them reach

a convenient point for making a clean backup of their persistent state on the volume. Once all the applications are ready, the system initializes the snapshot of the volume and then tells the applications that they can continue. The backup is made of the volume state at the point of the snapshot. And the applications were only blocked for a very short time rather than having to go offline for the duration of the backup.

Applications participate in the snapshot process, so the backup reflects a state that is easy to recover in case there is a future failure. Otherwise the backup might still be useful, but the state it captured would look more like the state if the system had crashed. Recovering from a system at the point of a crash can be more difficult or even impossible, since crashes occur at arbitrary times in the execution of the application. *Murphy's Law* says that crashes are most likely to occur at the worst possible time, that is, when the application data is in a state where recovery is impossible.

Another aspect of Windows is its support for asynchronous I/O. It is possible for a thread to start an I/O operation and then continue executing in parallel with the I/O. This feature is especially important on servers. There are various ways the thread can find out that the I/O has completed. One is to specify an event object at the time the call is made and then wait on it eventually. Another is to specify a queue to which a completion event will be posted by the system when the I/O is done. A third is to provide a callback procedure that the system calls when the I/O has completed. A fourth is to poll a location in memory that the I/O manager updates when the I/O completes.

The final aspect that we will mention is prioritized I/O. I/O priority is determined by the priority of the issuing thread, or it can be explicitly set. There are five priorities specified: *critical, high*, *normal*, *low*, and *very low*. Critical is reserved for the memory manager to avoid deadlocks that could otherwise occur when the system experiences extreme memory pressure. Low and very low priorities are used by background processes, like the disk defragmentation service and spyware scanners and desktop search, which are attempting to avoid interfering with normal operations of the system. Most I/O gets normal priority, but multimedia applications can mark their I/O as high to avoid glitches. Multimedia applications can alternatively use **bandwidth reservation** to request guaranteed bandwidth to access time-critical files, like music or video. The I/O system will provide the application with the optimal transfer size and the number of outstanding I/O operations that should be maintained to allow the I/O system to achieve the requested bandwidth guarantee.

11.7.2 Input/Output API Calls

The system call APIs provided by the I/O manager are not very different from those offered by most other operating systems. The basic operations are open, read, write, ioctl, and close, but there are also plug-and-play and power operations,

operations for setting parameters, as well as calls for flushing system buffers, and so on. At the Win32 layer these APIs are wrapped by interfaces that provide higher-level operations specific to particular devices. At the bottom, though, these wrappers open devices and perform these basic types of operations. Even some metadata operations, such as file rename, are implemented without specific system calls. They just use a special version of the ioctl operations. This will make more sense when we explain the implementation of I/O device stacks and the use of IRPs by the I/O manager.

I/O system call	Description
NtCreateFile	Open new or existing files or devices
NtReadFile	Read from a file or device
NtWriteFile	Write to a file or device
NtQueryDirectoryFile	Request information about a directory, including files
NtQueryVolumeInformationFile	Request information about a volume
NtSetVolumeInformationFile	Modify volume information
NtNotifyChangeDirectoryFile	Complete when any file in the directory or subtree is modified
NtQueryInformationFile	Request information about a file
NtSetInformationFile	Modify file information
NtLockFile	Lock a range of bytes in a file
NtUnlockFile	Remove a range lock
NtFsControlFile	Miscellaneous operations on a file
NtFlushBuffersFile	Flush in-memory file buffers to disk
NtCancelIoFile	Cancel outstanding I/O operations on a file
NtDeviceIoControlFile	Special operations on a device

Figure 11-35. Native NT API calls for performing I/O.

The native NT I/O system calls, in keeping with the general philosophy of Windows, take numerous parameters, and include many variations. Figure 11-35 lists the primary system-call interfaces to the I/O manager. NtCreateFile is used to open existing or new files. It provides security descriptors for new files, a rich description of the access rights requested, and gives the creator of new files some control over how blocks will be allocated. NtReadFile and NtWriteFile take a file handle, buffer, and length. They also take an explicit file offset, and allow a key to be specified for accessing locked ranges of bytes in the file. Most of the parameters are related to specifying which of the different methods to use for reporting completion of the (possibly asynchronous) I/O, as described above.

NtQueryDirectoryFile is an example of a standard paradigm in the executive where various Query APIs exist to access or modify information about specific types of objects. In this case, it is file objects that refer to directories. A parameter specifies what type of information is being requested, such as a list of the names in

the directory or detailed information about each file that is needed for an extended directory listing. Since this is really an I/O operation, all the standard ways of reporting that the I/O completed are supported. NtQueryVolumeInformationFile is like the directory query operation, but expects a file handle which represents an open volume which may or may not contain a file system. Unlike for directories, there are parameters than can be modified on volumes, and thus there is a separate API NtSetVolumeInformationFile.

NtNotifyChangeDirectoryFile is an example of an interesting NT paradigm. Threads can do I/O to determine whether any changes occur to objects (mainly file-system directories, as in this case, or registry keys). Because the I/O is asynchronous the thread returns and continues, and is only notified later when something is modified. The pending request is queued in the file system as an outstanding I/O operation using an I/O Request Packet. Notifications are problematic if you want to remove a file-system volume from the system, because the I/O operations are pending. So Windows supports facilities for canceling pending I/O operations, including support in the file system for forcibly dismounting a volume with pending I/O.

NtQueryInformationFile is the file-specific version of the system call for directories. It has a companion system call, NtSetInformationFile. These interfaces access and modify all sorts of information about file names, file features like encryption and compression and sparseness, and other file attributes and details, including looking up the internal file id or assigning a unique binary name (object id) to a file.

These system calls are essentially a form of ioctl specific to files. The set operation can be used to rename or delete a file. But note that they take handles, not file names, so a file first must be opened before being renamed or deleted. They can also be used to rename the alternative data streams on NTFS (see Sec. 11.8).

Separate APIs, NtLockFile and NtUnlockFile, exist to set and remove byte-range locks on files. NtCreateFile allows access to an entire file to be restricted by using a sharing mode. An alternative is these lock APIs, which apply mandatory access restrictions to a range of bytes in the file. Reads and writes must supply a *key* matching the key provided to NtLockFile in order to operate on the locked ranges.

Similar facilities exist in UNIX, but there it is discretionary whether applications heed the range locks. NtFsControlFile is much like the preceding Query and Set operations, but is a more generic operation aimed at handling file-specific operations that do not fit within the other APIs. For example, some operations are specific to a particular file system.

Finally, there are miscellaneous calls such as NtFlushBuffersFile. Like the UNIX sync call, it forces file-system data to be written back to disk. NtCancelIoFile cancels outstanding I/O requests for a particular file, and NtDeviceIoControlFile implements ioctl operations for devices. The list of operations is actually much longer. There are system calls for deleting files by name, and for querying

the attributes of a specific file—but these are just wrappers around the other I/O manager operations we have listed and did not really need to be implemented as separate system calls. There are also system calls for dealing with **I/O completion ports**, a queuing facility in Windows that helps multithreaded servers make efficient use of asynchronous I/O operations by readying threads by demand and reducing the number of context switches required to service I/O on dedicated threads.

11.7.3 Implementation of I/O

The Windows I/O system consists of the plug-and-play services, the device power manager, the I/O manager, and the device-driver model. Plug-and-play detects changes in hardware configuration and builds or tears down the device stacks for each device, as well as causing the loading and unloading of device drivers. The device power manager adjusts the power state of the I/O devices to reduce system power consumption when devices are not in use. The I/O manager provides support for manipulating I/O kernel objects, and IRP-based operations like IoCallDrivers and IoCompleteRequest. But most of the work required to support Windows I/O is implemented by the device drivers themselves.

Device Drivers

To make sure that device drivers work well with the rest of Windows, Microsoft has defined the **WDM** (**Windows Driver Model**) that device drivers are expected to conform with. The **WDK** (**Windows Driver Kit**) contains documentation and examples to help developers produce drivers which conform to the WDM. Most Windows drivers start out as copies of an appropriate sample driver from the WDK, which is then modified by the driver writer.

Microsoft also provides a **driver verifier** which validates many of the actions of drivers to be sure that they conform to the WDM requirements for the structure and protocols for I/O requests, memory management, and so on. The verifier ships with the system, and administrators can control it by running *verifier.exe*, which allows them to configure which drivers are to be checked and how extensive (i.e., expensive) the checks should be.

Even with all the support for driver development and verification, it is still very difficult to write even simple drivers in Windows, so Microsoft has built a system of wrappers called the **WDF** (**Windows Driver Foundation**) that runs on top of WDM and simplifies many of the more common requirements, mostly related to correct interaction with device power management and plug-and-play operations.

To further simplify driver writing, as well as increase the robustness of the system, WDF includes the **UMDF** (**User-Mode Driver Framework**) for writing drivers as services that execute in processes. And there is the **KMDF** (**Kernel-Mode**

Driver Framework) for writing drivers as services that execute in the kernel, but with many of the details of WDM made automagical. Since underneath it is the WDM that provides the driver model, that is what we will focus on in this section.

Devices in Windows are represented by device objects. Device objects are also used to represent hardware, such as buses, as well as software abstractions like file systems, network protocol engines, and kernel extensions, such as antivirus filter drivers. All these are organized by producing what Windows calls a *device stack*, as previously shown in Fig. 11-14.

I/O operations are initiated by the I/O manager calling an executive API IoCallDriver with pointers to the top device object and to the IRP representing the I/O request. This routine finds the driver object associated with the device object. The operation types that are specified in the IRP generally correspond to the I/O manager system calls described above, such as create, read, and close.

Figure 11-36 shows the relationships for a single level of the device stack. For each of these operations a driver must specify an entry point. IoCallDriver takes the operation type out of the IRP, uses the device object at the current level of the device stack to find the driver object, and indexes into the driver dispatch table with the operation type to find the corresponding entry point into the driver. The driver is then called and passed the device object and the IRP.

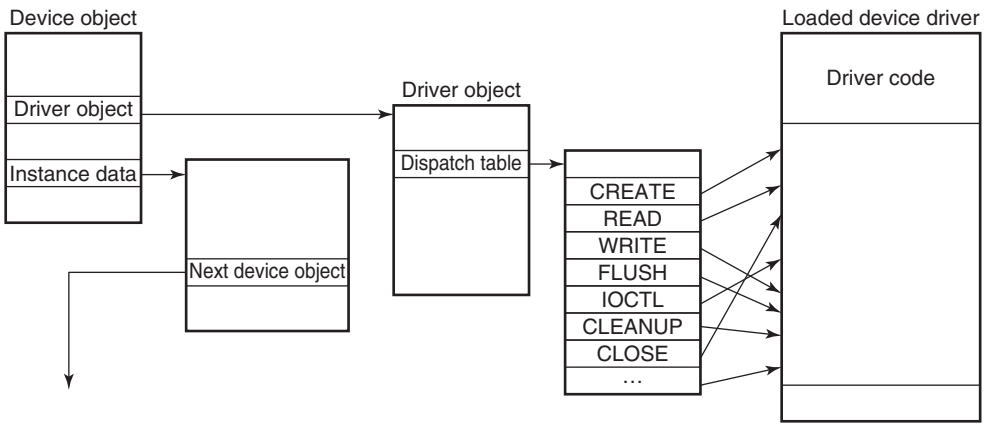

Figure 11-36. A single level in a device stack.

Once a driver has finished processing the request represented by the IRP, it has three options. It can call IoCallDriver again, passing the IRP and the next device object in the device stack. It can declare the I/O request to be completed and return to its caller. Or it can queue the IRP internally and return to its caller, having declared that the I/O request is still pending. This latter case results in an asynchronous I/O operation, at least if all the drivers above in the stack agree and also return to their callers.

I/O Request Packets

Figure 11-37 shows the major fields in the IRP. The bottom of the IRP is a dynamically sized array containing fields that can be used by each driver for the device stack handling the request. These *stack* fields also allow a driver to specify the routine to call when completing an I/O request. During completion each level of the device stack is visited in reverse order, and the completion routine assigned by each driver is called in turn. At each level the driver can continue to complete the request or decide there is still more work to do and leave the request pending, suspending the I/O completion for the time being.

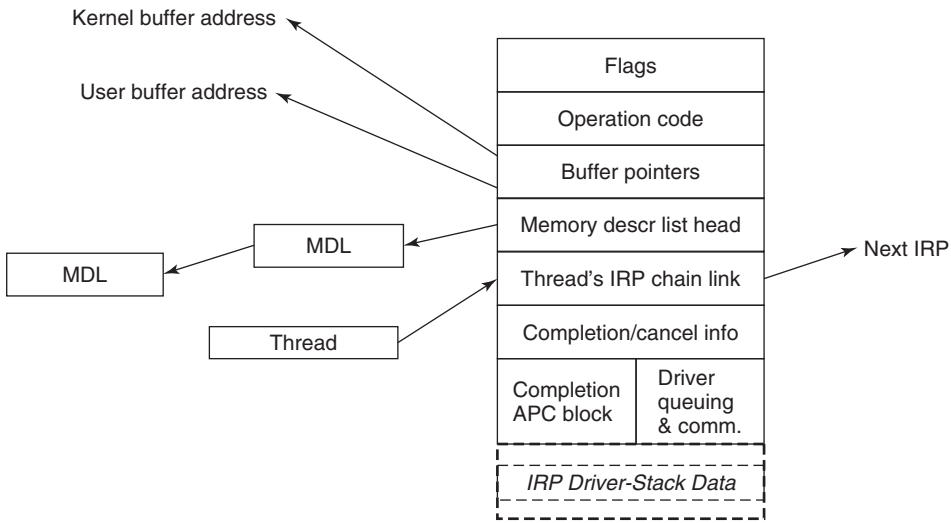

Figure 11-37. The major fields of an I/O Request Packet.

When allocating an IRP, the I/O manager has to know how deep the particular device stack is so that it can allocate a sufficiently large IRP. It keeps track of the stack depth in a field in each device object as the device stack is formed. Note that there is no formal definition of what the next device object is in any stack. That information is held in private data structures belonging to the previous driver on the stack. In fact, the stack does not really have to be a stack at all. At any layer a driver is free to allocate new IRPs, continue to use the original IRP, send an I/O operation to a different device stack, or even switch to a system worker thread to continue execution.

The IRP contains flags, an operation code for indexing into the driver dispatch table, buffer pointers for possibly both kernel and user buffers, and a list of **MDLs** (**Memory Descriptor Lists**) which are used to describe the physical pages represented by the buffers, that is, for DMA operations. There are fields used for cancellation and completion operations. The fields in the IRP that are used to queue the

IRP to devices while it is being processed are reused when the I/O operation has finally completed to provide memory for the APC control object used to call the I/O manager's completion routine in the context of the original thread. There is also a link field used to link all the outstanding IRPs to the thread that initiated them.

Device Stacks

A driver in Windows may do all the work by itself, as the printer driver does in Fig. 11-38. On the other hand, drivers may also be stacked, which means that a request may pass through a sequence of drivers, each doing part of the work. Two stacked drivers are also illustrated in Fig. 11-38.

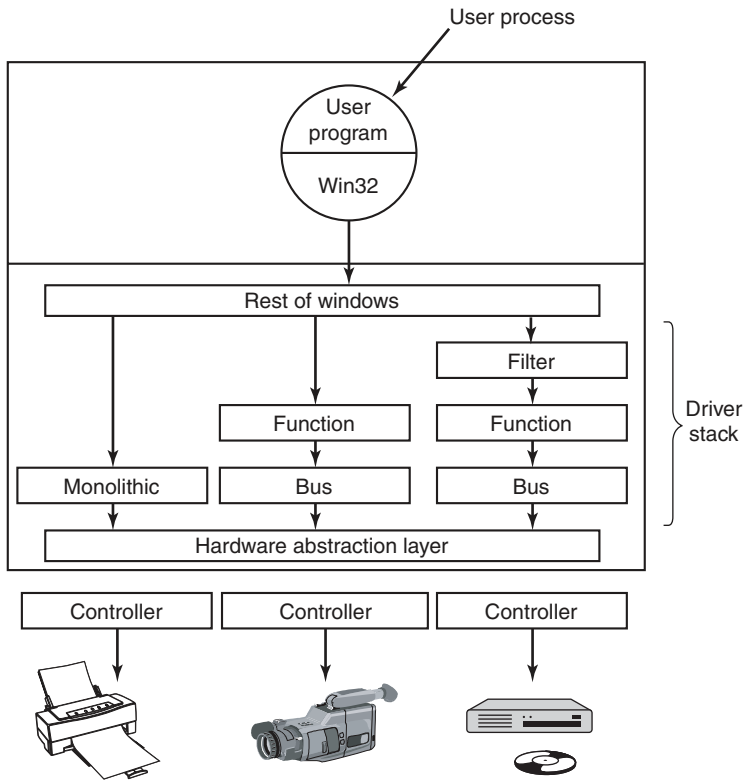

Figure 11-38. Windows allows drivers to be stacked to work with a specific instance of a device. The stacking is represented by device objects.

One common use for stacked drivers is to separate the bus management from the functional work of controlling the device. Bus management on the PCI bus is quite complicated on account of many kinds of modes and bus transactions. By

separating this work from the device-specific part, driver writers are freed from learning how to control the bus. They can just use the standard bus driver in their stack. Similarly, USB and SCSI drivers have a device-specific part and a generic part, with common drivers being supplied by Windows for the generic part.

Another use of stacking drivers is to be able to insert **filter drivers** into the stack. We have already looked at the use of file-system filter drivers, which are inserted above the file system. Filter drivers are also used for managing physical hardware. A filter driver performs some transformation on the operations as the IRP flows down the device stack, as well as during the completion operation with the IRP flows back up through the completion routines each driver specified. For example, a filter driver could compress data on the way to the disk or encrypt data on the way to the network. Putting the filter here means that neither the application program nor the true device driver has to be aware of it, and it works automatically for all data going to (or coming from) the device.

Kernel-mode device drivers are a serious problem for the reliability and stability of Windows. Most of the kernel crashes in Windows are due to bugs in device drivers. Because kernel-mode device drivers all share the same address space with the kernel and executive layers, errors in the drivers can corrupt system data structures, or worse. Some of these bugs are due to the astonishingly large numbers of device drivers that exist for Windows, or to the development of drivers by less-experienced system programmers. The bugs are also due to the enormous amount of detail involved in writing a correct driver for Windows.

The I/O model is powerful and flexible, but all I/O is fundamentally asynchronous, so race conditions can abound. Windows 2000 added the plug-and-play and device power management facilities from the Win9x systems to the NT-based Windows for the first time. This put a large number of requirements on drivers to deal correctly with devices coming and going while I/O packets are in the middle of being processed. Users of PCs frequently dock/undock devices, close the lid and toss notebooks into briefcases, and generally do not worry about whether the little green activity light happens to still be on. Writing device drivers that function correctly in this environment can be very challenging, which is why WDF was developed to simplify the Windows Driver Model.

Many books are available about the Windows Driver Model and the newer Windows Driver Foundation (Kanetkar, 2008; Orwick & Smith, 2007; Reeves, 2010; Viscarola et al., 2007; and Vostokov, 2009).

11.8 THE WINDOWS NT FILE SYSTEM

Windows supports several file systems, the most important of which are **FAT-16**, **FAT-32**, and **NTFS** (**NT File System**). FAT-16 is the old MS-DOS file system. It uses 16-bit disk addresses, which limits it to disk partitions no larger than 2 GB. Mostly it is used to access floppy disks, for those customers that still

use them. FAT-32 uses 32-bit disk addresses and supports disk partitions up to 2 TB. There is no security in FAT-32 and today it is really used only for transportable media, like flash drives. NTFS is the file system developed specifically for the NT version of Windows. Starting with Windows XP it became the default file system installed by most computer manufacturers, greatly improving the security and functionality of Windows. NTFS uses 64-bit disk addresses and can (theoretically) support disk partitions up to 2^{64} bytes, although other considerations limit it to smaller sizes.

In this chapter we will examine the NTFS file system because it is a modern one with many interesting features and design innovations. It is large and complex and space limitations prevent us from covering all of its features, but the material presented below should give a reasonable impression of it.

11.8.1 Fundamental Concepts

Individual file names in NTFS are limited to 255 characters; full paths are limited to 32,767 characters. File names are in Unicode, allowing people in countries not using the Latin alphabet (e.g., Greece, Japan, India, Russia, and Israel) to write file names in their native language. For example, *φιλε* is a perfectly legal file name. NTFS fully supports case-sensitive names (so *foo* is different from *Foo* and *FOO*). The Win32 API does not support case-sensitivity fully for file names and not at all for directory names. The support for case sensitivity exists when running the POSIX subsystem in order to maintain compatibility with UNIX. Win32 is not case sensitive, but it is case preserving, so file names can have different case letters in them. Though case sensitivity is a feature that is very familiar to users of UNIX, it is largely inconvenient to ordinary users who do not make such distinctions normally. For example, the Internet is largely case-insensitive today.

An NTFS file is not just a linear sequence of bytes, as FAT-32 and UNIX files are. Instead, a file consists of multiple attributes, each represented by a stream of bytes. Most files have a few short streams, such as the name of the file and its 64-bit object ID, plus one long (unnamed) stream with the data. However, a file can also have two or more (long) data streams as well. Each stream has a name consisting of the file name, a colon, and the stream name, as in *foo:stream1*. Each stream has its own size and is lockable independently of all the other streams. The idea of multiple streams in a file is not new in NTFS. The file system on the Apple Macintosh uses two streams per file, the data fork and the resource fork. The first use of multiple streams for NTFS was to allow an NT file server to serve Macintosh clients. Multiple data streams are also used to represent metadata about files, such as the thumbnail pictures of JPEG images that are available in the Windows GUI. But alas, the multiple data streams are fragile and frequently fall off files when they are transported to other file systems, transported over the network, or even when backed up and later restored, because many utilities ignore them.

NTFS is a hierarchical file system, similar to the UNIX file system. The separator between component names is "\", however, instead of "/", a fossil inherited from the compatibility requirements with CP/M when MS-DOS was created (CP/M used the slash for flags). Unlike UNIX the concept of the current working directory, hard links to the current directory (.) and the parent directory (..) are implemented as conventions rather than as a fundamental part of the file-system design. Hard links are supported, but used only for the POSIX subsystem, as is NTFS support for traversal checking on directories (the 'x' permission in UNIX).

Symbolic links in are supported for NTFS. Creation of symbolic links is normally restricted to administrators to avoid security issues like spoofing, as UNIX experienced when symbolic links were first introduced in 4.2BSD. The implementation of symbolic links uses an NTFS feature called **reparse points** (discussed later in this section). In addition, compression, encryption, fault tolerance, journaling, and sparse files are also supported. These features and their implementations will be discussed shortly.

11.8.2 Implementation of the NT File System

NTFS is a highly complex and sophisticated file system that was developed specifically for NT as an alternative to the HPFS file system that had been developed for OS/2. While most of NT was designed on dry land, NTFS is unique among the components of the operating system in that much of its original design took place aboard a sailboat out on the Puget Sound (following a strict protocol of work in the morning, beer in the afternoon). Below we will examine a number of features of NTFS, starting with its structure, then moving on to file-name lookup, file compression, journaling, and file encryption.

File System Structure

Each NTFS volume (e.g., disk partition) contains files, directories, bitmaps, and other data structures. Each volume is organized as a linear sequence of blocks (clusters in Microsoft's terminology), with the block size being fixed for each volume and ranging from 512 bytes to 64 KB, depending on the volume size. Most NTFS disks use 4-KB blocks as a compromise between large blocks (for efficient transfers) and small blocks (for low internal fragmentation). Blocks are referred to by their offset from the start of the volume using 64-bit numbers.

The principal data structure in each volume is the **MFT** (**Master File Table**), which is a linear sequence of fixed-size 1-KB records. Each MFT record describes one file or one directory. It contains the file's attributes, such as its name and timestamps, and the list of disk addresses where its blocks are located. If a file is extremely large, it is sometimes necessary to use two or more MFT records to contain the list of all the blocks, in which case the first MFT record, called the **base record**, points to the additional MFT records. This overflow scheme dates back to

CP/M, where each directory entry was called an extent. A bitmap keeps track of which MFT entries are free.

The MFT is itself a file and as such can be placed anywhere within the volume, thus eliminating the problem with defective sectors in the first track. Furthermore, the file can grow as needed, up to a maximum size of 2^{48} records.

The MFT is shown in Fig. 11-39. Each MFT record consists of a sequence of (attribute header, value) pairs. Each attribute begins with a header telling which attribute this is and how long the value is. Some attribute values are variable length, such as the file name and the data. If the attribute value is short enough to fit in the MFT record, it is placed there. If it is too long, it is placed elsewhere on the disk and a pointer to it is placed in the MFT record. This makes NTFS very efficient for small files, that is, those that can fit within the MFT record itself.

The first 16 MFT records are reserved for NTFS metadata files, as illustrated in Fig. 11-39. Each record describes a normal file that has attributes and data blocks, just like any other file. Each of these files has a name that begins with a dollar sign to indicate that it is a metadata file. The first record describes the MFT file itself. In particular, it tells where the blocks of the MFT file are located so that the system can find the MFT file. Clearly, Windows needs a way to find the first block of the MFT file in order to find the rest of the file-system information. The way it finds the first block of the MFT file is to look in the boot block, where its address is installed when the volume is formatted with the file system.

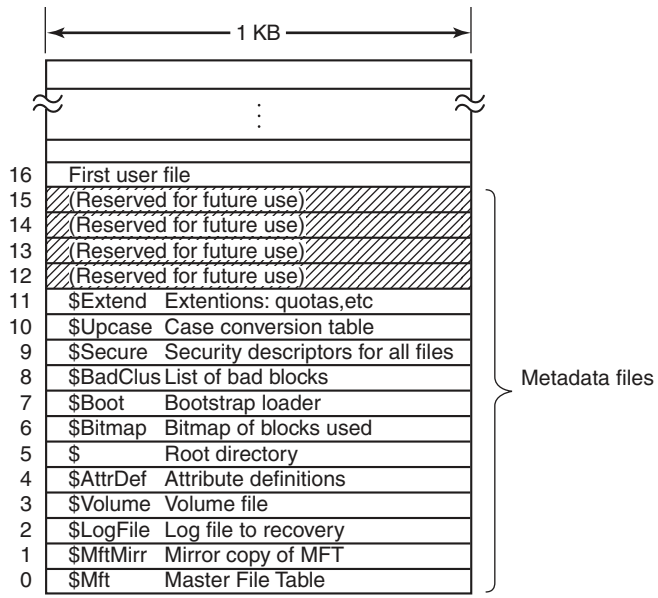

Figure 11-39. The NTFS master file table.

Record 1 is a duplicate of the early portion of the MFT file. This information is so precious that having a second copy can be critical in the event one of the first blocks of the MFT ever becomes unreadable. Record 2 is the log file. When structural changes are made to the file system, such as adding a new directory or removing an existing one, the action is logged here before it is performed, in order to increase the chance of correct recovery in the event of a failure during the operation, such as a system crash. Changes to file attributes are also logged here. In fact, the only changes not logged here are changes to user data. Record 3 contains information about the volume, such as its size, label, and version.

As mentioned above, each MFT record contains a sequence of (attribute header, value) pairs. The *$AttrDef* file is where the attributes are defined. Information about this file is in MFT record 4. Next comes the root directory, which itself is a file and can grow to arbitrary length. It is described by MFT record 5.

Free space on the volume is kept track of with a bitmap. The bitmap is itself a file, and its attributes and disk addresses are given in MFT record 6. The next MFT record points to the bootstrap loader file. Record 8 is used to link all the bad blocks together to make sure they never occur in a file. Record 9 contains the security information. Record 10 is used for case mapping. For the Latin letters A-Z case mapping is obvious (at least for people who speak Latin). Case mapping for other languages, such as Greek, Armenian, or Georgian (the country, not the state), is less obvious to Latin speakers, so this file tells how to do it. Finally, record 11 is a directory containing miscellaneous files for things like disk quotas, object identifiers, reparse points, and so on. The last four MFT records are reserved for future use.

Each MFT record consists of a record header followed by the (attribute header, value) pairs. The record header contains a magic number used for validity checking, a sequence number updated each time the record is reused for a new file, a count of references to the file, the actual number of bytes in the record used, the identifier (index, sequence number) of the base record (used only for extension records), and some other miscellaneous fields.

NTFS defines 13 attributes that can appear in MFT records. These are listed in Fig. 11-40. Each attribute header identifies the attribute and gives the length and location of the value field along with a variety of flags and other information. Usually, attribute values follow their attribute headers directly, but if a value is too long to fit in the MFT record, it may be put in separate disk blocks. Such an attribute is said to be a **nonresident attribute**. The data attribute is an obvious candidate. Some attributes, such as the name, may be repeated, but all attributes must appear in a fixed order in the MFT record. The headers for resident attributes are 24 bytes long; those for nonresident attributes are longer because they contain information about where to find the attribute on disk.

The standard information field contains the file owner, security information, the timestamps needed by POSIX, the hard-link count, the read-only and archive bits, and so on. It is a fixed-length field and is always present. The file name is a

Attribute	Description
Standard information	Flag bits, timestamps, etc.
File name	File name in Unicode; may be repeated for MS-DOS name
Security descriptor	Obsolete. Security information is now in $Extend$Secure
Attribute list	Location of additional MFT records, if needed
Object ID	64-bit file identifier unique to this volume
Reparse point	Used for mounting and symbolic links
Volume name	Name of this volume (used only in $Volume)
Volume information	Volume version (used only in $Volume)
Index root	Used for directories
Index allocation	Used for very large directories
Bitmap	Used for very large directories
Logged utility stream	Controls logging to $LogFile
Data	Stream data; may be repeated

Figure 11-40. The attributes used in MFT records.

variable-length Unicode string. In order to make files with non–MS-DOS names accessible to old 16-bit programs, files can also have an 8 + 3 MS-DOS **short name**. If the actual file name conforms to the MS-DOS 8 + 3 naming rule, a secondary MS-DOS name is not needed.

In NT 4.0, security information was put in an attribute, but in Windows 2000 and later, security information all goes into a single file so that multiple files can share the same security descriptions. This results in significant savings in space within most MFT records and in the file system overall because the security info for so many of the files owned by each user is identical.

The attribute list is needed in case the attributes do not fit in the MFT record. This attribute then tells where to find the extension records. Each entry in the list contains a 48-bit index into the MFT telling where the extension record is and a 16-bit sequence number to allow verification that the extension record and base records match up.

NTFS files have an ID associated with them that is like the i-node number in UNIX. Files can be opened by ID, but the IDs assigned by NTFS are not always useful when the ID must be persisted because it is based on the MFT record and can change if the record for the file moves (e.g., if the file is restored from backup). NTFS allows a separate object ID attribute which can be set on a file and never needs to change. It can be kept with the file if it is copied to a new volume, for example.

The reparse point tells the procedure parsing the file name that it has do something special. This mechanism is used for explicitly mounting file systems and for symbolic links. The two volume attributes are used only for volume identification.

The next three attributes deal with how directories are implemented. Small ones are just lists of files but large ones are implemented using B+ trees. The logged utility stream attribute is used by the encrypting file system.

Finally, we come to the attribute that is the most important of all: the data stream (or in some cases, streams). An NTFS file has one or more data streams associated with it. This is where the payload is. The **default data stream** is unnamed (i.e., *dirpath\file name::$DATA*), but the **alternate data streams** each have a name, for example, *dirpath\file name:streamname:$DATA*.

For each stream, the stream name, if present, goes in this attribute header. Following the header is either a list of disk addresses telling which blocks the stream contains, or for streams of only a few hundred bytes (and there are many of these), the stream itself. Putting the actual stream data in the MFT record is called an **immediate file** (Mullender and Tanenbaum, 1984).

Of course, most of the time the data does not fit in the MFT record, so this attribute is usually nonresident. Let us now take a look at how NTFS keeps track of the location of nonresident attributes, in particular data.

Storage Allocation

The model for keeping track of disk blocks is that they are assigned in runs of consecutive blocks, where possible, for efficiency reasons. For example, if the first logical block of a stream is placed in block 20 on the disk, then the system will try hard to place the second logical block in block 21, the third logical block in 22, and so on. One way to achieve these runs is to allocate disk storage several blocks at a time, when possible.

The blocks in a stream are described by a sequence of records, each one describing a sequence of logically contiguous blocks. For a stream with no holes in it, there will be only one such record. Streams that are written in order from beginning to end all belong in this category. For a stream with one hole in it (e.g., only blocks 0–49 and blocks 60–79 are defined), there will be two records. Such a stream could be produced by writing the first 50 blocks, then seeking forward to logical block 60 and writing another 20 blocks. When a hole is read back, all the missing bytes are zeros. Files with holes are called **sparse files**.

Each record begins with a header giving the offset of the first block within the stream. Next comes the offset of the first block not covered by the record. In the example above, the first record would have a header of (0, 50) and would provide the disk addresses for these 50 blocks. The second one would have a header of (60, 80) and would provide the disk addresses for these 20 blocks.

Each record header is followed by one or more pairs, each giving a disk address and run length. The disk address is the offset of the disk block from the start of its partition; the run length is the number of blocks in the run. As many pairs as needed can be in the run record. Use of this scheme for a three-run, nine-block stream is illustrated in Fig. 11-41.

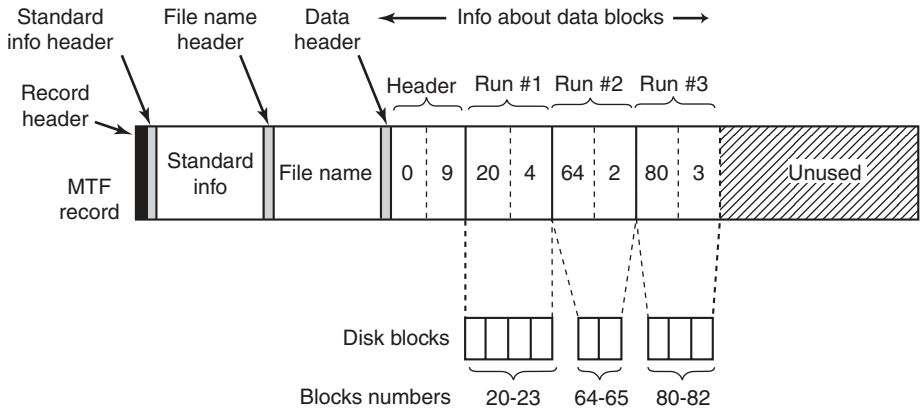

Figure 11-41. An MFT record for a three-run, nine-block stream.

In this figure we have an MFT record for a short stream of nine blocks (header 0–8). It consists of the three runs of consecutive blocks on the disk. The first run is blocks 20–23, the second is blocks 64–65, and the third is blocks 80–82. Each of these runs is recorded in the MFT record as a (disk address, block count) pair. How many runs there are depends on how well the disk block allocator did in finding runs of consecutive blocks when the stream was created. For an n-block stream, the number of runs can be anything from 1 through n.

Several comments are worth making here. First, there is no upper limit to the size of streams that can be represented this way. In the absence of address compression, each pair requires two 64-bit numbers in the pair for a total of 16 bytes. However, a pair could represent 1 million or more consecutive disk blocks. In fact, a 20-MB stream consisting of 20 separate runs of 1 million 1-KB blocks each fits easily in one MFT record, whereas a 60-KB stream scattered into 60 isolated blocks does not.

Second, while the straightforward way of representing each pair takes 2×8 bytes, a compression method is available to reduce the size of the pairs below 16. Many disk addresses have multiple high-order zero-bytes. These can be omitted. The data header tells how many are omitted, that is, how many bytes are actually used per address. Other kinds of compression are also used. In practice, the pairs are often only 4 bytes.

Our first example was easy: all the file information fit in one MFT record. What happens if the file is so large or highly fragmented that the block information does not fit in one MFT record? The answer is simple: use two or more MFT records. In Fig. 11-42 we see a file whose base record is in MFT record 102. It has too many runs for one MFT record, so it computes how many extension records it needs, say, two, and puts their indices in the base record. The rest of the record is used for the first k data runs.

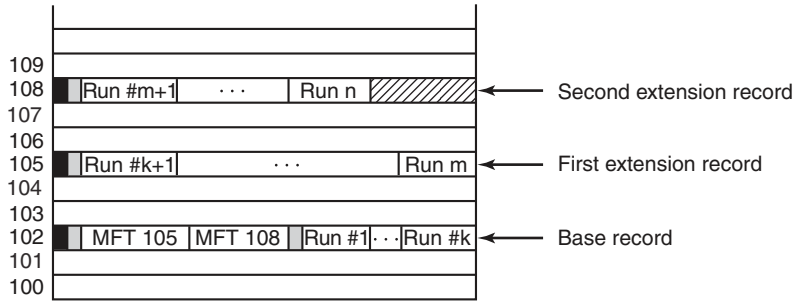

Figure 11-42. A file that requires three MFT records to store all its runs.

Note that Fig. 11-42 contains some redundancy. In theory, it should not be necessary to specify the end of a sequence of runs because this information can be calculated from the run pairs. The reason for "overspecifying" this information is to make seeking more efficient: to find the block at a given file offset, it is necessary to examine only the record headers, not the run pairs.

When all the space in record 102 has been used up, storage of the runs continues with MFT record 105. As many runs are packed in this record as fit. When this record is also full, the rest of the runs go in MFT record 108. In this way, many MFT records can be used to handle large fragmented files.

A problem arises if so many MFT records are needed that there is no room in the base MFT to list all their indices. There is also a solution to this problem: the list of extension MFT records is made nonresident (i.e., stored in other disk blocks instead of in the base MFT record). Then it can grow as large as needed.

An MFT entry for a small directory is shown in Fig. 11-43. The record contains a number of directory entries, each of which describes one file or directory. Each entry has a fixed-length structure followed by a variable-length file name. The fixed part contains the index of the MFT entry for the file, the length of the file name, and a variety of other fields and flags. Looking for an entry in a directory consists of examining all the file names in turn.

Large directories use a different format. Instead, of listing the files linearly, a B+ tree is used to make alphabetical lookup possible and to make it easy to insert new names in the directory in the proper place.

The NTFS parsing of the path *\foo\bar* begins at the root directory for *C:*, whose blocks can be found from entry 5 in the MFT (see Fig. 11-39). The string "foo" is looked up in the root directory, which returns the index into the MFT for the directory *foo*. This directory is then searched for the string "bar", which refers to the MFT record for this file. NTFS performs access checks by calling back into the security reference monitor, and if everything is cool it searches the MFT record for the attribute *::$DATA*, which is the default data stream.

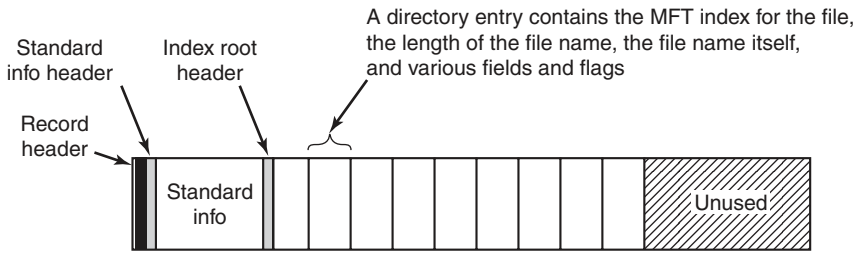

Figure 11-43. The MFT record for a small directory.

We now have enough information to finish describing how file-name lookup occurs for a file *\??\C:\foo\bar*. In Fig. 11-20 we saw how the Win32, the native NT system calls, and the object and I/O managers cooperated to open a file by sending an I/O request to the NTFS device stack for the *C:* volume. The I/O request asks NTFS to fill in a file object for the remaining path name, *\foo\bar*.

Having found file *bar*, NTFS will set pointers to its own metadata in the file object passed down from the I/O manager. The metadata includes a pointer to the MFT record, information about compression and range locks, various details about sharing, and so on. Most of this metadata is in data structures shared across all file objects referring to the file. A few fields are specific only to the current open, such as whether the file should be deleted when it is closed. Once the open has succeeded, NTFS calls IoCompleteRequest to pass the IRP back up the I/O stack to the I/O and object managers. Ultimately a handle for the file object is put in the handle table for the current process, and control is passed back to user mode. On subsequent ReadFile calls, an application can provide the handle, specifying that this file object for *C:\foo\bar* should be included in the read request that gets passed down the *C:* device stack to NTFS.

In addition to regular files and directories, NTFS supports hard links in the UNIX sense, and also symbolic links using a mechanism called **reparse points**. NTFS supports tagging a file or directory as a reparse point and associating a block of data with it. When the file or directory is encountered during a file-name parse, the operation fails and the block of data is returned to the object manager. The object manager can interpret the data as representing an alternative path name and then update the string to parse and retry the I/O operation. This mechanism is used to support both symbolic links and mounted file systems, redirecting the search to a different part of the directory hierarchy or even to a different partition.

Reparse points are also used to tag individual files for file-system filter drivers. In Fig. 11-20 we showed how file-system filters can be installed between the I/O manager and the file system. I/O requests are completed by calling IoCompleteRequest, which passes control to the completion routines each driver represented

in the device stack inserted into the IRP as the request was being made. A driver that wants to tag a file associates a reparse tag and then watches for completion requests for file open operations that failed because they encountered a reparse point. From the block of data that is passed back with the IRP, the driver can tell if this is a block of data that the driver itself has associated with the file. If so, the driver will stop processing the completion and continue processing the original I/O request. Generally, this will involve proceeding with the open request, but there is a flag that tells NTFS to ignore the reparse point and open the file.

File Compression

NTFS supports transparent file compression. A file can be created in compressed mode, which means that NTFS automatically tries to compress the blocks as they are written to disk and automatically uncompresses them when they are read back. Processes that read or write compressed files are completely unaware that compression and decompression are going on.

Compression works as follows. When NTFS writes a file marked for compression to disk, it examines the first 16 (logical) blocks in the file, irrespective of how many runs they occupy. It then runs a compression algorithm on them. If the resulting compressed data can be stored in 15 or fewer blocks, they are written to the disk, preferably in one run, if possible. If the compressed data still take 16 blocks, the 16 blocks are written in uncompressed form. Then blocks 16–31 are examined to see if they can be compressed to 15 blocks or fewer, and so on.

Figure 11-44(a) shows a file in which the first 16 blocks have successfully compressed to eight blocks, the second 16 blocks failed to compress, and the third 16 blocks have also compressed by 50%. The three parts have been written as three runs and stored in the MFT record. The "missing" blocks are stored in the MFT entry with disk address 0 as shown in Fig. 11-44(b). Here the header (0, 48) is followed by five pairs, two for the first (compressed) run, one for the uncompressed run, and two for the final (compressed) run.

When the file is read back, NTFS has to know which runs are compressed and which ones are not. It can tell based on the disk addresses. A disk address of 0 indicates that it is the final part of 16 compressed blocks. Disk block 0 may not be used for storing data, to avoid ambiguity. Since block 0 on the volume contains the boot sector, using it for data is impossible anyway.

Random access to compressed files is actually possible, but tricky. Suppose that a process does a seek to block 35 in Fig. 11-44. How does NTFS locate block 35 in a compressed file? The answer is that it has to read and decompress the entire run first. Then it knows where block 35 is and can pass it to any process that reads it. The choice of 16 blocks for the compression unit was a compromise. Making it shorter would have made the compression less effective. Making it longer would have made random access more expensive.

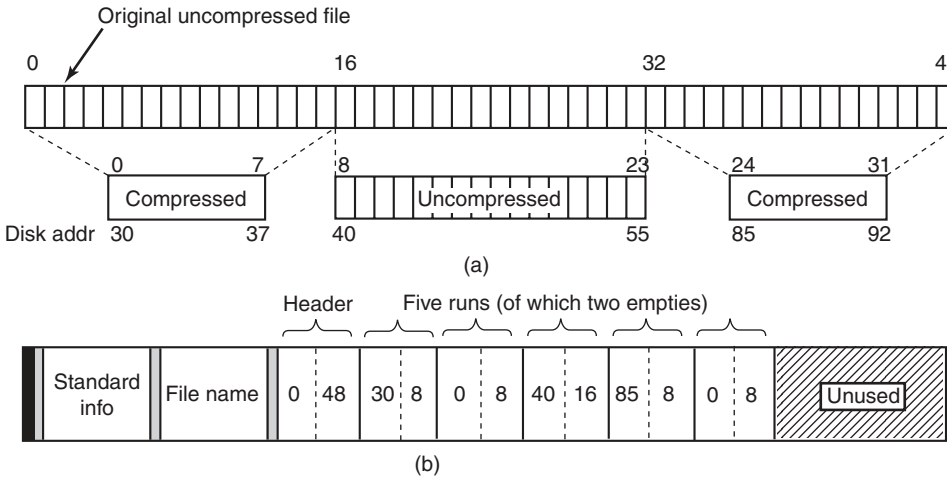

Figure 11-44. (a) An example of a 48-block file being compressed to 32 blocks. (b) The MFT record for the file after compression.

Journaling

NTFS supports two mechanisms for programs to detect changes to files and directories. First is an operation, NtNotifyChangeDirectoryFile, that passes a buffer and returns when a change is detected to a directory or directory subtree. The result is that the buffer has a list of *change records*. If it is too small, records are lost.

The second mechanism is the NTFS change journal. NTFS keeps a list of all the change records for directories and files on the volume in a special file, which programs can read using special file-system control operations, that is, the *FSCTL_QUERY_USN_JOURNAL* option to the NtFsControlFile API. The journal file is normally very large, and there is little likelihood that entries will be reused before they can be examined.

File Encryption

Computers are used nowadays to store all kinds of sensitive data, including plans for corporate takeovers, tax information, and love letters, which the owners do not especially want revealed to anyone. Information loss can happen when a notebook computer is lost or stolen, a desktop system is rebooted using an MS-DOS floppy disk to bypass Windows security, or a hard disk is physically removed from one computer and installed on another one with an insecure operating system.

Windows addresses these problems by providing an option to encrypt files, so that even in the event the computer is stolen or rebooted using MS-DOS, the files will be unreadable. The normal way to use Windows encryption is to mark certain directories as encrypted, which causes all the files in them to be encrypted, and

new files moved to them or created in them to be encrypted as well. The actual encryption and decryption are not managed by NTFS itself, but by a driver called **EFS** (**Encryption File System**), which registers callbacks with NTFS.

EFS provides encryption for specific files and directories. There is also another encryption facility in Windows called **BitLocker** which encrypts almost all the data on a volume, which can help protect data no matter what—as long as the user takes advantage of the mechanisms available for strong keys. Given the number of systems that are lost or stolen all the time, and the great sensitivity to the issue of identity theft, making sure secrets are protected is very important. An amazing number of notebooks go missing every day. Major Wall Street companies supposedly average losing one notebook per week in taxicabs in New York City alone.

11.9 WINDOWS POWER MANAGEMENT

The **power manager** rides herd on power usage throughout the system. Historically management of power consumption consisted of shutting off the monitor display and stopping the disk drives from spinning. But the issue is rapidly becoming more complicated due to requirements for extending how long notebooks can run on batteries, and energy-conservation concerns related to desktop computers being left on all the time and the high cost of supplying power to the huge server farms that exist today.

Newer power-management facilities include reducing the power consumption of components when the system is not in use by switching individual devices to standby states, or even powering them off completely using *soft* power switches. Multiprocessors shut down individual CPUs when they are not needed, and even the clock rates of the running CPUs can be adjusted downward to reduce power consumption. When a processor is idle, its power consumption is also reduced since it needs to do nothing except wait for an interrupt to occur.

Windows supports a special shut down mode called **hibernation**, which copies all of physical memory to disk and then reduces power consumption to a small trickle (notebooks can run weeks in a hibernated state) with little battery drain. Because all the memory state is written to disk, you can even replace the battery on a notebook while it is hibernated. When the system resumes after hibernation it restores the saved memory state (and reinitializes the I/O devices). This brings the computer back into the same state it was before hibernation, without having to login again and start up all the applications and services that were running. Windows optimizes this process by ignoring unmodified pages backed by disk already and compressing other memory pages to reduce the amount of I/O bandwidth required. The hibernation algorithm automatically tunes itself to balance between I/O and processor throughput. If there is more processor available, it uses expensive but more effective compression to reduce the I/O bandwidth needed. When I/O bandwidth is sufficient, hibernation will skip the compression altogether. With

the current generation of multiprocessors, both hibernation and resume can be performed in a few seconds even on systems with many gigabytes of RAM.

An alternative to hibernation is **standby mode** where the power manager reduces the entire system to the lowest power state possible, using just enough power to the refresh the dynamic RAM. Because memory does not need to be copied to disk, this is somewhat faster than hibernation on some systems.

Despite the availability of hibernation and standby, many users are still in the habit of shutting down their PC when they finish working. Windows uses hibernation to perform a pseudo shutdown and startup, called *HiberBoot*, that is much faster than normal shutdown and startup. When the user tells the system to shutdown, HiberBoot logs the user off and then hibernates the system at the point they would normally login again. Later, when the user turns the system on again, HiberBoot will resume the system at the login point. To the user it looks like shutdown was very, very fast because most of the system initialization steps are skipped. Of course, sometimes the system needs to perform a real shutdown in order to fix a problem or install an update to the kernel. If the system is told to reboot rather than shutdown, the system undergoes a real shutdown and performs a normal boot.

On phones and tablets, as well as the newest generation of laptops, computing devices are expected to be always on yet consume little power. To provide this experience Modern Windows implements a special version of power management called **CS** (**connected standby**). CS is possible on systems with special networking hardware which is able to listen for traffic on a small set of connections using much less power than if the CPU were running. A CS system always appears to be on, coming out of CS as soon as the screen is turned on by the user. Connected standby is different than the regular standby mode because a CS system will also come out of standby when it receives a packet on a monitored connection. Once the battery begins to run low, a CS system will go into the hibernation state to avoid completely exhausting the battery and perhaps losing user data.

Achieving good battery life requires more than just turning off the processor as often as possible. It is also important to keep the processor off as long as possible. The CS network hardware allows the processors to stay off until data have arrived, but other events can also cause the processors to be turned back on. In NT-based Windows device drivers, system services, and the applications themselves frequently run for no particular reason other than to *check on things*. Such *polling* activity is usually based on setting timers to periodically run code in the system or application. Timer-based polling can produce a cacophony of events turning on the processor. To avoid this, Modern Windows requires that timers specify an imprecision parameter which allows the operating system to coalesce timer events and reduce the number of separate occasions one of the processors will have to be turned back on. Windows also formalizes the conditions under which an application that is not actively running can execute code in the background. Operations like checking for updates or freshening content cannot be performed solely by requesting to run when a timer expires. An application must defer to the operating system about

when to run such background activities. For example, checking for updates might occur only once a day or at the next time the device is charging its battery. A set of system brokers provide a variety of conditions which can be used to limit when background activity is performed. If a background task needs to access a low-cost network or utilize a user's credentials, the brokers will not execute the task until the requisite conditions are present.

Many applications today are implemented with both local code and services in the cloud. Windows provides WNS (*Windows Notification Service*) which allows third-party services to push notifications to a Windows device in CS without requiring the CS network hardware to specifically listen for packets from the third party's servers. WNS notifications can signal time-critical events, such as the arrival of a text message or a VoIP call. When a WNS packet arrives, the processor will have to be turned on to process it, but the ability of the CS network hardware to discriminate between traffic from different connections means the processor does not have to awaken for every random packet that arrives at the network interface.

11.10 SECURITY IN WINDOWS 8

NT was originally designed to meet the U.S. Department of Defense's C2 security requirements (DoD 5200.28-STD), the Orange Book, which secure DoD systems must meet. This standard requires operating systems to have certain properties in order to be classified as secure enough for certain kinds of military work. Although Windows was not specifically designed for C2 compliance, it inherits many security properties from the original security design of NT, including the following:

1. Secure login with antispoofing measures.
2. Discretionary access controls.
3. Privileged access controls.
4. Address-space protection per process.
5. New pages must be zeroed before being mapped in.
6. Security auditing.

Let us review these items briefly

Secure login means that the system administrator can require all users to have a password in order to log in. Spoofing is when a malicious user writes a program that displays the login prompt or screen and then walks away from the computer in the hope that an innocent user will sit down and enter a name and password. The name and password are then written to disk and the user is told that login has

failed. Windows prevents this attack by instructing users to hit CTRL-ALT-DEL to log in. This key sequence is always captured by the keyboard driver, which then invokes a system program that puts up the genuine login screen. This procedure works because there is no way for user processes to disable CTRL-ALT-DEL processing in the keyboard driver. But NT can and does disable use of the CTRL-ALT-DEL secure attention sequence in some cases, particularly for consumers and in systems that have accessibility for the disabled enabled, on phones, tablets, and the Xbox, where there rarely is a physical keyboard.

Discretionary access controls allow the owner of a file or other object to say who can use it and in what way. Privileged access controls allow the system administrator (superuser) to override them when needed. Address-space protection simply means that each process has its own protected virtual address space not accessible by any unauthorized process. The next item means that when the process heap grows, the pages mapped in are initialized to zero so that processes cannot find any old information put there by the previous owner (hence the zeroed page list in Fig. 11-34, which provides a supply of zeroed pages for this purpose). Finally, security auditing allows the administrator to produce a log of certain security-related events.

While the Orange Book does not specify what is to happen when someone steals your notebook computer, in large organizations one theft a week is not unusual. Consequently, Windows provides tools that a conscientious user can use to minimize the damage when a notebook is stolen or lost (e.g., secure login, encrypted files, etc.). Of course, conscientious users are precisely the ones who do not lose their notebooks—it is the others who cause the trouble.

In the next section we will describe the basic concepts behind Windows security. After that we will look at the security system calls. Finally, we will conclude by seeing how security is implemented.

11.10.1 Fundamental Concepts

Every Windows user (and group) is identified by an **SID** (**Security ID**). SIDs are binary numbers with a short header followed by a long random component. Each SID is intended to be unique worldwide. When a user starts up a process, the process and its threads run under the user's SID. Most of the security system is designed to make sure that each object can be accessed only by threads with authorized SIDs.

Each process has an **access token** that specifies an SID and other properties. The token is normally created by *winlogon*, as described below. The format of the token is shown in Fig. 11-45. Processes can call GetTokenInformation to acquire this information. The header contains some administrative information. The expiration time field could tell when the token ceases to be valid, but it is currently not used. The *Groups* field specifies the groups to which the process belongs, which is needed for the POSIX subsystem. The default **DACL** (**Discretionary ACL**) is the

access control list assigned to objects created by the process if no other ACL is specified. The user SID tells who owns the process. The restricted SIDs are to allow untrustworthy processes to take part in jobs with trustworthy processes but with less power to do damage.

Finally, the privileges listed, if any, give the process special powers denied ordinary users, such as the right to shut the machine down or access files to which access would otherwise be denied. In effect, the privileges split up the power of the superuser into several rights that can be assigned to processes individually. In this way, a user can be given some superuser power, but not all of it. In summary, the access token tells who owns the process and which defaults and powers are associated with it.

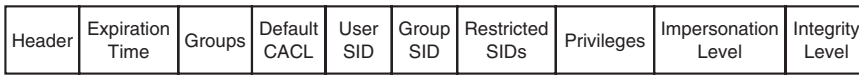

Figure 11-45. Structure of an access token.

When a user logs in, *winlogon* gives the initial process an access token. Subsequent processes normally inherit this token on down the line. A process' access token initially applies to all the threads in the process. However, a thread can acquire a different access token during execution, in which case the thread's access token overrides the process' access token. In particular, a client thread can pass its access rights to a server thread to allow the server to access the client's protected files and other objects. This mechanism is called **impersonation**. It is implemented by the transport layers (i.e., ALPC, named pipes, and TCP/IP) and used by RPC to communicate from clients to servers. The transports use internal interfaces in the kernel's security reference monitor component to extract the security context for the current thread's access token and ship it to the server side, where it is used to construct a token which can be used by the server to impersonate the client.

Another basic concept is the **security descriptor**. Every object has a security descriptor associated with it that tells who can perform which operations on it. The security descriptors are specified when the objects are created. The NTFS file system and the registry maintain a persistent form of security descriptor, which is used to create the security descriptor for File and Key objects (the object-manager objects representing open instances of files and keys).

A security descriptor consists of a header followed by a DACL with one or more **ACEs** (**Access Control Entries**). The two main kinds of elements are Allow and Deny. An Allow element specifies an SID and a bitmap that specifies which operations processes that SID may perform on the object. A Deny element works the same way, except a match means the caller may not perform the operation. For example, Ida has a file whose security descriptor specifies that everyone has read access, Elvis has no access. Cathy has read/write access, and Ida herself has full

access. This simple example is illustrated in Fig. 11-46. The SID Everyone refers to the set of all users, but it is overridden by any explicit ACEs that follow.

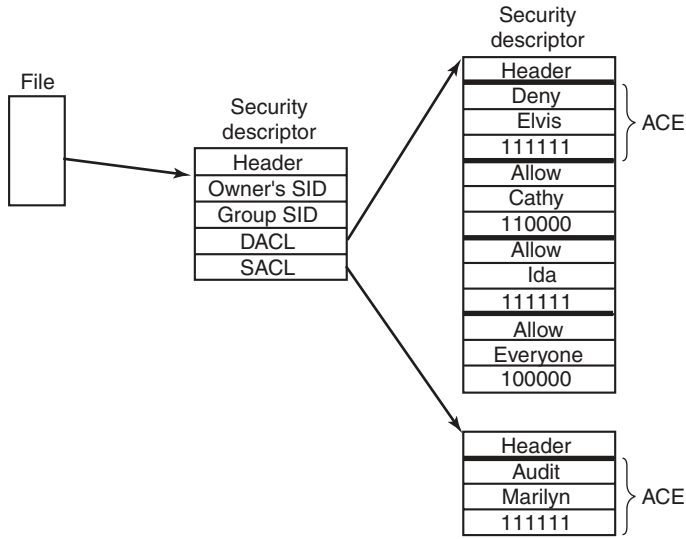

Figure 11-46. An example security descriptor for a file.

In addition to the DACL, a security descriptor also has a **SACL** (**System Access Control list**), which is like a DACL except that it specifies not who may use the object, but which operations on the object are recorded in the systemwide security event log. In Fig. 11-46, every operation that Marilyn performs on the file will be logged. The SACL also contains the **integrity level**, which we will describe shortly.

11.10.2 Security API Calls

Most of the Windows access-control mechanism is based on security descriptors. The usual pattern is that when a process creates an object, it provides a security descriptor as one of the parameters to the CreateProcess, CreateFile, or other object-creation call. This security descriptor then becomes the security descriptor attached to the object, as we saw in Fig. 11-46. If no security descriptor is provided in the object-creation call, the default security in the caller's access token (see Fig. 11-45) is used instead.

Many of the Win32 API security calls relate to the management of security descriptors, so we will focus on those here. The most important calls are listed in Fig. 11-47. To create a security descriptor, storage for it is first allocated and then