liferation of threads packages made writing portable code difficult. Eventually, the system calls used to manage threads were standardized as part of POSIX (P1003.1c).

The POSIX specification did not take a position on whether threads should be implemented in the kernel or in user space. The advantage of having user-space threads is that they can be implemented without having to change the kernel and thread switching is very efficient. The disadvantage of user-space threads is that if one thread blocks (e.g., on I/O, a semaphore, or a page fault), all the threads in the process block because the kernel thinks there is only one thread and does not schedule the process until the blocking thread is released. Thus the calls defined in P1003.1c were carefully chosen to be implementable either way. As long as user programs adhere carefully to the P1003.1c semantics, both implementations should work correctly. The most commonly-used thread calls are listed in Fig. 10-8. When kernel threads are used, these calls are true system calls; when user threads are used, these calls are implemented entirely in a user-space run-time library.

(For the truly alert reader, note that we have a typographical problem now. If the kernel manages threads, then calls such as "pthread_create," are system calls and following our convention should be set in Helvetica, like this: pthread_create. However, if they are simply user-space library calls, our convention for all procedure names is to use Times Italics, like this: *pthread_create*. Without prejudice, we will simply use Helvetica, also in the next chapter, in which it is never clear which Win32 API calls are really system calls. It could be worse: in the Algol 68 Report there was a period that changed the grammar of the language slightly when printed in the wrong font.)

Thread call	Description
pthread_create	Create a new thread in the caller's address space
pthread_exit	Terminate the calling thread
pthread_join	Wait for a thread to terminate
pthread_mutex_init	Create a new mutex
pthread_mutex_destroy	Destroy a mutex
pthread_mutex_lock	Lock a mutex
pthread_mutex_unlock	Unlock a mutex
pthread_cond_init	Create a condition variable
pthread_cond_destroy	Destroy a condition variable
pthread_cond_wait	Wait on a condition variable
pthread_cond_signal	Release one thread waiting on a condition variable

Figure 10-8. The principal POSIX thread calls.

Let us briefly examine the thread calls shown in Fig. 10-8. The first call, pthread_create, creates a new thread. It is called by

```
err = pthread_create(&tid, attr, function, arg);
```

This call creates a new thread in the current process running the code *function* with *arg* passed to it as a parameter. The new thread's ID is stored in memory at the location pointed to by the first parameters. The *attr* parameter can be used to specify certain attributes for the new thread, such as its scheduling priority. After successful completion, one more thread is running in the caller's address space than was before the call.

A thread that has done its job and wants to terminate calls pthread_exit. A thread can wait for another thread to exit by calling pthread_join. If the thread waited for has already exited, the pthread_join finishes immediately. Otherwise it blocks.

Threads can synchronize using locks called **mutexes**. Typically a mutex guards some resource, such as a buffer shared by two threads. To make sure that only one thread at a time accesses the shared resource, threads are expected to lock the mutex before touching the resource and unlock it when they are done. As long as all threads obey this protocol, race conditions can be avoided. Mutexes are like binary semaphores, that is, semaphores that can take on only the values of 0 and 1. The name "mutex" comes from the fact that mutexes are used to ensure mutual exclusion on some resource.

Mutexes can be created and destroyed by the calls pthread_mutex_init and pthread_mutex_destroy, respectively. A mutex can be in one of two states: locked or unlocked. When a thread needs to set a lock on an unlocked mutex (using pthread_mutex_lock), the lock is set and the thread continues. However, when a thread tries to lock a mutex that is already locked, it blocks. When the locking thread is finished with the shared resource, it is expected to unlock the corresponding mutex by calling pthread_mutex_unlock.

Mutexes are intended for short-term locking, such as protecting a shared variable. They are not intended for long-term synchronization, such as waiting for a tape drive to become free. For long-term synchronization, **condition variables** are provided. These are created and destroyed by calls to pthread_cond_init and pthread_cond_destroy, respectively.

A condition variable is used by having one thread wait on it, and another thread signal it. For example, having discovered that the tape drive it needs is busy, a thread would do pthread_cond_wait on a condition variable that all the threads have agreed to associate with the tape drive. When the thread using the tape drive is finally done with it (possibly even hours later), it uses pthread_cond_signal to release exactly one thread waiting on that condition variable (if any). If no thread is waiting, the signal is lost. In other words, condition variables do not count like semaphores. A few other operations are also defined on threads, mutexes, and condition variables.

10.3.3 Implementation of Processes in UNIX

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

The kernel maintains two key data structures related to processes, the **process table** and the **user structure**. The process table is resident all the time and contains information needed for all processes, even those that are not currently present in memory. The user structure is swapped or paged out when its associated process is not in memory, in order not to waste memory on information that is not needed.

The information in the process table falls into the following broad categories:

1. **Scheduling parameters**. Process priority, amount of CPU time consumed recently, amount of time spent sleeping recently. Together, these are used to determine which process to run next.

2. **Memory image**. Pointers to the text, data, and stack segments, or, if paging is used, to their page tables. If the text segment is shared, the text pointer points to the shared text table. When the process is not in memory, information about how to find its parts on disk is here too.

3. **Signals**. Masks showing which signals are being ignored, which are being caught, which are being temporarily blocked, and which are in the process of being delivered.

4. **Miscellaneous**. Current process state, event being waited for, if any, time until alarm clock goes off, PID, PID of the parent process, and user and group identification.

The user structure contains information that is not needed when the process is not physically in memory and runnable. For example, although it is possible for a process to be sent a signal while it is swapped out, it is not possible for it to read a file. For this reason, information about signals must be in the process table, so they are in memory all the time, even when the process is not present in memory. On the other hand, information about file descriptors can be kept in the user structure and brought in only when the process is in memory and runnable.

The information contained in the user structure includes the following items:

1. **Machine registers**. When a trap to the kernel occurs, the machine registers (including the floating-point ones, if used) are saved here.
2. **System call state**. Information about the current system call, including the parameters, and results.
3. **File descriptor table**. When a system call involving a file descriptor is invoked, the file descriptor is used as an index into this table to locate the in-core data structure (i-node) corresponding to this file.
4. **Accounting**. Pointer to a table that keeps track of the user and system CPU time used by the process. Some systems also maintain limits here on the amount of CPU time a process may use, the maximum size of its stack, the number of page frames it may consume, and other items.
5. **Kernel stack**. A fixed stack for use by the kernel part of the process.

Bearing the use of these tables in mind, it is now easy to explain how processes are created in UNIX. When a fork system call is executed, the calling process traps to the kernel and looks for a free slot in the process table for use by the child. If it finds one, it copies all the information from the parent's process table entry to the child's entry. It then allocates memory for the child's data and stack segments, and makes exact copies of the parent's data and stack segments there. The user structure (which is often kept adjacent to the stack segment), is copied along with the stack. The text segment may either be copied or shared since it is read only. At this point, the child is ready to run.

When a user types a command, say, ls on the terminal, the shell creates a new process by forking off a clone of itself. The new shell then calls exec to overlay its memory with the contents of the executable file *ls*. The steps involved are shown in Fig. 10-9.

The mechanism for creating a new process is actually fairly straightforward. A new process table slot and user area are created for the child process and filled in largely from the parent. The child is given a PID, its memory map is set up, and it is given shared access to its parent's files. Then its registers are set up and it is ready to run.

In principle, a complete copy of the address space should be made, since the semantics of fork say that no memory is shared between parent and child. However, copying memory is expensive, so all modern UNIX systems cheat. They give the child its own page tables, but have them point to the parent's pages, only marked read only. Whenever the child tries to write on a page, it gets a protection fault. The kernel sees this and then allocates a new copy of the page to the child and marks it read/write. In this way, only pages that are actually written have to

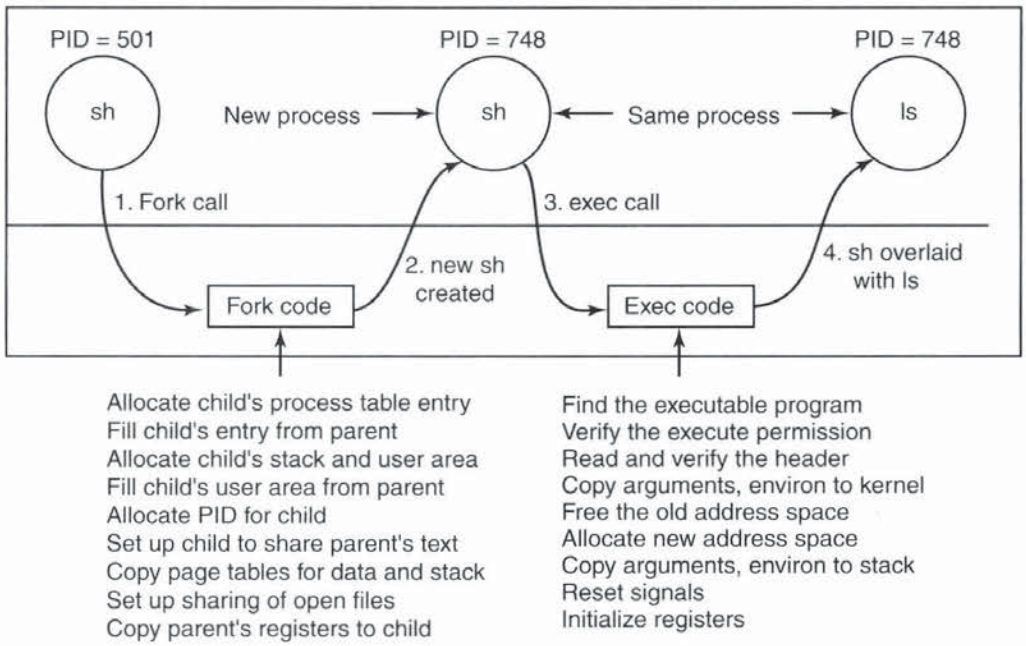


Figure 10-9. The steps in executing the command *ls* typed to the shell.

be copied. This mechanism is called **copy-on-write**. It has the additional benefit of not requiring two copies of the program in memory, thus saving RAM.

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

Now the new address space must be created and filled in. If the system supports mapped files, as System V, BSD, and most other UNIX systems do, the new page tables are set up to indicate that no pages are in memory, except perhaps one stack page, but that the address space is backed by the executable file on disk. When the new process starts running, it will immediately get a page fault, which will cause the first page of code to be paged in from the executable file. In this way, nothing has to be loaded in advance, so programs can start quickly and fault in just those pages they need and no more. Finally, the arguments and environment strings are copied to the new stack, the signals are reset, and the registers are initialized to all zeros. At this point, the new command can start running.

Threads in UNIX

The implementation of threads depends on whether they are supported in the kernel or not. If they are not, such as in 4BSD, the implementation is entirely in a user-space library. If they are, as in System V and Solaris, the kernel has some

work to do. We discussed threads in a general way in Chap. 2. Here we will just make a few remarks about kernel threads in UNIX.

The main issue in introducing threads is maintaining the correct traditional UNIX semantics. First consider fork. Suppose that a process with multiple (kernel) threads does a fork system call. Should all the other threads be created in the new process? For the moment, let us answer that question with yes. Suppose that one of the other threads was blocked reading from the keyboard. Should the corresponding thread in the new process also be blocked reading from the keyboard? If so, which one gets the next line typed? If not, what should that thread be doing in the new process? The same problem holds for many other things threads can do. In a single-threaded process, the problem does not arise because the one and only thread cannot be blocked when calling fork. Now consider the case that the other threads are not created in the child process. Suppose that one of the not-created threads holds a mutex that the one-and-only thread in the new process tries to acquire after doing the fork. The mutex will never be released and the one thread will hang forever. Numerous other problems exist too. There is no simple solution.

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

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

Threads in Linux

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

The heart of the Linux implementation of threads is a new system call, clone, that is not present in any other version of UNIX. It is called as follows:

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

The call creates a new thread, either in the current process or in a new process, depending on *sharing_flags*. If the new thread is in the current process, it shares the address space with existing threads and every subsequent write to any byte in the address space by any thread is immediately visible to all the other threads in

the process. On the other hand, if the address space is not shared, then the new thread gets an exact copy of the address space, but subsequent writes by the new thread are not visible to the old ones. These semantics are the same as fork.

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

The *sharing_flags* parameter is a bitmap that allows a much finer grain of sharing than traditional UNIX systems. Five bits are defined, as listed in Fig. 10-10. Each bit controls some aspect of sharing, and each of the bits can be set independently of the other ones. The *CLONE_VM* bit determines whether the virtual memory (i.e., address space) is shared with the old threads or copied. If it is set, the new thread just moves in with the existing ones, so the clone call effectively creates a new thread in an existing process. If the bit is cleared, the new thread gets its own address space. Having its own address space means that the effect of its STORE instructions are not visible to the existing threads. This behavior is similar to fork, except as noted below. Creating a new address space is effectively the definition of a new process.

Flag	Meaning when set	Meaning when cleared
CLONE_VM	Create a new thread	Create a new process
CLONE_FS	Share umask, root, and working dirs	Do not share them
CLONE_FILES	Share the file descriptors	Copy the file descriptors
CLONE_SIGHAND	Share the signal handler table	Copy the table
CLONE_PID	New thread gets old PID	New thread gets own PID

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

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

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

its parent's PID. This feature is needed during system booting. User processes are not permitted to enable it.

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

Scheduling in UNIX

Let us now examine the UNIX scheduling algorithm. Because UNIX has always been a multiprogramming system, its scheduling algorithm was designed from the beginning to provide good response to interactive processes. It is a two-level algorithm. The low-level algorithm picks the process to run next from the set of processes in memory and ready to run. The high-level algorithm moves processes between memory and disk so that all processes get a chance to be in memory and run.

Each version of UNIX has a slightly different low-level scheduling algorithm, but most of them are close to the generic one we will describe now. The low-level algorithm uses multiple queues. Each queue is associated with a range of nonoverlapping priority values. Processes executing in user mode (the top of the iceberg) have positive values. Processes executing in kernel mode (doing system calls) have negative values. Negative values have the highest priority and large positive values have the lowest, as illustrated in Fig. 10-11. Only processes that are in memory and ready to run are located on the queues, since the choice must be made from this set.

When the (low-level) scheduler runs, it searches the queues starting at the highest priority (i.e., most negative value) until it finds a queue that is occupied. The first process on that queue is then chosen and started. It is allowed to run for a maximum of one quantum, typically 100 msec, or until it blocks. If a process uses up its quantum, it is put back on the end of its queue, and the scheduling algorithm is run again. Thus processes within the same priority range share the CPU using a round-robin algorithm.

Once a second, each process' priority is recalculated according to a formula involving three components:

$$priority = CPU_usage + nice + base$$

Based on its new priority, each process is attached to the appropriate queue of Fig. 10-11, usually by dividing the priority by a constant to get the queue number. Let us now briefly examine each of the three components of the priority formula.

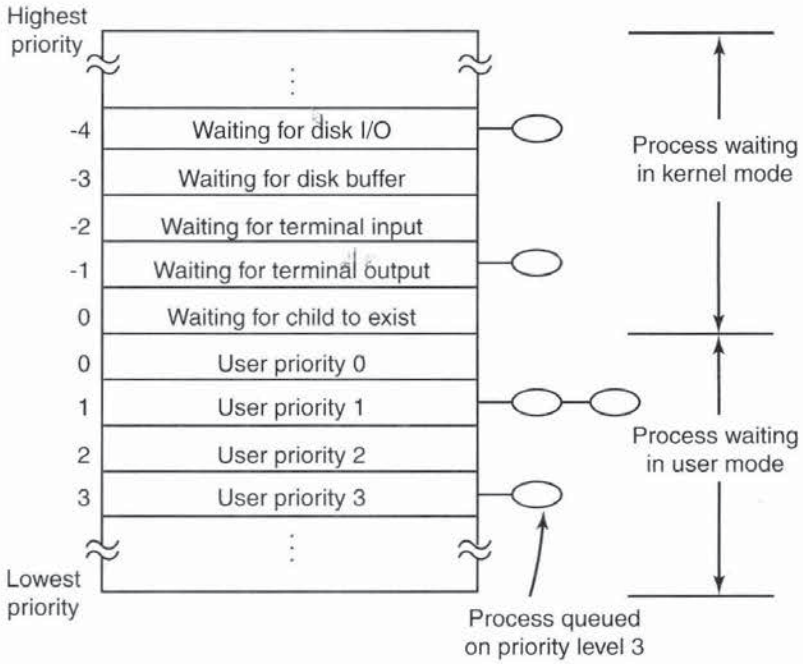


Figure 10-11. The UNIX scheduler is based on a multilevel queue structure.

CPU_usage, represents the average number of clock ticks per second that the process has had during the past few seconds. Every time the clock ticks, the CPU usage counter in the running process' process table entry is incremented by 1. This counter will ultimately be added to the process' priority giving it a higher numerical value and thus putting it on a lower-priority queue.

However, UNIX does not punish a process forever for using the CPU, so *CPU_usage* decays with time. Different versions of UNIX do the decay slightly differently. One way that has been used is to add the current value of *CPU_usage* to the number of ticks acquired in the past ΔT and divide the result by 2. This algorithm weights the most recent ΔT by ½, the one before that by ¼, and so on. This weighting algorithm is very fast because it just has one addition and one shift, but other weighting schemes have also been used.

Every process has a *nice* value associated with it. The default value is 0, but the allowed range is generally −20 to +20. A process can set *nice* to a value in the range 0 to 20 by the nice system call. A user computing π to a billion places in the background might put this call in his program to be nice to the other users. Only the system administrator may ask for *better* than normal service (meaning values from −20 to −1). Deducing the reason for this rule is left as an exercise for the reader.

When a process traps to the kernel to make a system call, it is entirely possible that the process has to block before completing the system call and returning to user mode. For example, it may have just done a waitpid system call and have

to wait for one of its children to exit. It may also have to wait for terminal input or for disk I/O to complete, to mention only a few of the many possibilities. When it blocks, it is removed from the queue structure, since it is unable to run.

However, when the event it was waiting for occurs, it is put onto a queue with a negative value. The choice of queue is determined by the event it was waiting for. In Fig. 10-11, disk I/O is shown as having the highest priority, so a process that has just read or written a block from the disk will probably get the CPU within 100 msec. The relative priority of disk I/O, terminal I/O, etc. is hardwired into the operating system, and can only be modified by changing some constants in the source code and recompiling the system. These (negative) values are represented by *base* in the formula given above and are spaced far enough apart that processes being restarted for different reasons are clearly separated into different queues.

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

Scheduling in Linux

Scheduling is one of the few areas in which Linux uses a different algorithm from UNIX. We have just examined the UNIX scheduling algorithm, so we will now look at the Linux algorithm. To start with, Linux threads are kernel threads, so scheduling is based on threads, not processes. Linux distinguishes three classes of threads for scheduling purposes:

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

Real-time FIFO threads are the highest priority and are not preemptable except by a newly-readied real-time FIFO thread. Real-time round-robin threads are the same as real-time FIFO threads except that they are preemptable by the clock. If multiple real-time round-robin threads are ready, each one is run for its quantum, after which it goes to the end of the list of real-time round-robin threads. Neither of these classes is actually real time in any sense. Deadlines cannot be specified and guarantees are not given. These classes are simply higher priority than threads in the standard timesharing class. The reason Linux calls them real time

is that Linux is conformant to the P1003.4 standard ("real-time" extensions to UNIX) which uses those names.

Each thread has a scheduling priority. The default value is 20, but that can be altered using the nice(value) system call to a value of $20 - value$. Since *value* must be in the range −20 to +19, priorities always fall in the range: $1 \leq priority \leq 40$. The intention is that the quality of service is roughly proportional to the priority, with higher priority threads getting faster response time and a larger fraction of the CPU time than lower priority threads.

In addition to a priority, each thread has a quantum associated with it. The quantum is the number of clock ticks the thread may continue to run for. The clock runs at 100 Hz by default, so each tick is 10 msec, which is called a **jiffy**. The scheduler uses the priority and quantum as follows. It first computes the **goodness** of each ready thread by applying the following rules:

```
if (class == real_time) goodness = 1000 + priority;
if (class == timesharing && quantum > 0) goodness = quantum + priority;
if (class == timesharing && quantum == 0) goodness = 0;
```

Both real-time classes count for the first rule. All that marking a thread as real time does is make sure it gets a higher goodness than all timesharing threads. The algorithm has one little extra feature: if the process that ran last still has some quantum left, it gets a bonus point, so that it wins any ties. The idea here is that all things being equal, it is more efficient to run the previous process since its pages and cache blocks are likely to be loaded.

Given this background, the scheduling algorithm is very simple: when a scheduling decision is made, the thread with the highest goodness is selected. As the selected thread runs, at every clock tick, its quantum is decremented by 1. The CPU is taken away from a thread if any of these conditions occur:

1. Its quantum hits 0.
2. The thread blocks on I/O, a semaphore, or something else.
3. A previously blocked thread with a higher goodness becomes ready.

Since the quanta keep counting down, sooner or later every ready thread will grind its quantum into the ground and they will all be 0. However, I/O bound threads that are currently blocked may have some quantum left. At this point the scheduler resets the quantum of *all* threads, ready and blocked, using the rule:

```
quantum = (quantum/2) + priority
```

where the new quantum is in jiffies. A thread that is highly compute bound will usually exhaust its quantum quickly and have it 0 when quanta are reset, giving it a quantum equal to its priority. An I/O-bound thread may have considerable quantum left and thus get a larger quantum next time. If nice is not used, the priority will be 20, so the quantum becomes 20 jiffies or 200 msec. On the other

hand, for a highly I/O bound thread, it may still have a quantum of 20 left when quanta are reset, so if its priority is 20, its new quantum becomes $20/2 + 20 = 30$ jiffies. If another reset happens before it has spent 1 tick, next time it gets a quantum of $30/2 + 20 = 35$ jiffies. The asymptotic value in jiffies is twice the priority. As a consequence of this algorithm, I/O-bound threads get larger quanta and thus higher goodness than compute-bound threads. This gives I/O-bound threads preference in scheduling.

Another property of this algorithm is that when compute-bound threads are competing for the CPU, ones with higher priority get a larger fraction of it. To see this, consider two compute-bound threads, *A*, with priority 20 and *B*, with priority 5. *A* goes first and 20 ticks later has used up its quantum. Then *B* gets to run for 5 quanta. At this point the quanta are reset. *A* gets 20 and *B* gets 5. This goes on forever, so *A* is getting 80% of the CPU and *B* is getting 20% of the CPU.

10.3.4 Booting UNIX

The exact details of how UNIX is booted vary from system to system. Below we will look briefly at how 4.4BSD is booted, but the ideas are somewhat similar for all versions. When the computer starts, the first sector of the boot disk (the master boot record) is read into memory and executed. This sector contains a small (512-byte) program that loads a standalone program called *boot* from the boot device, usually an IDE or SCSI disk. The *boot* program first copies itself to a fixed high memory address to free up low memory for the operating system.

Once moved, *boot* reads the root directory of the boot device. To do this, it must understand the file system and directory format, which it does. Then it reads in the operating system kernel and jumps to it. At this point, *boot* has finished its job and the kernel is running.

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

The C code also has considerable initialization to do, but this is more logical than physical. It starts out by allocating a message buffer to help debug boot problems. As initialization proceeds, messages are written here about what is happening, so they can be fished out after a boot failure by a special diagnostic program. Think of this as the operating system's cockpit flight recorder (the black box investigators look for after a plane crash).

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

At this point the system begins autoconfiguration. Using configuration files telling what kinds of I/O devices might be present, it begins probing the devices to

see which ones actually are present. If a probed device responds to the probe, it is added to a table of attached devices. If it fails to respond, it is assumed to be absent and ignored henceforth.

Once the device list has been determined, the device drivers must be located. This is one area in which UNIX systems differ somewhat. In particular, 4.4BSD cannot load device drivers dynamically, so any I/O device whose driver was not statically linked with the kernel cannot be used. In contrast, some other versions of UNIX, such as Linux, can load drivers dynamically (as can all versions of MS-DOS and Windows, incidentally).

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

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

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

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

```
login:
```

on the terminal's screen and tries to read the user's name from the keyboard. When someone sits down at the terminal and provides a login name, *getty* terminates by executing */bin/login*, the login program. *Login* then asks for a

password, encrypts it, and verifies it against the encrypted password stored in the password file, */etc/passwd*. If it is correct, *login* replaces itself with the user's shell, which then waits for the first command. If it is incorrect, *login* just asks for another user name. This mechanism is illustrated in Fig. 10-12 for a system with three terminals.

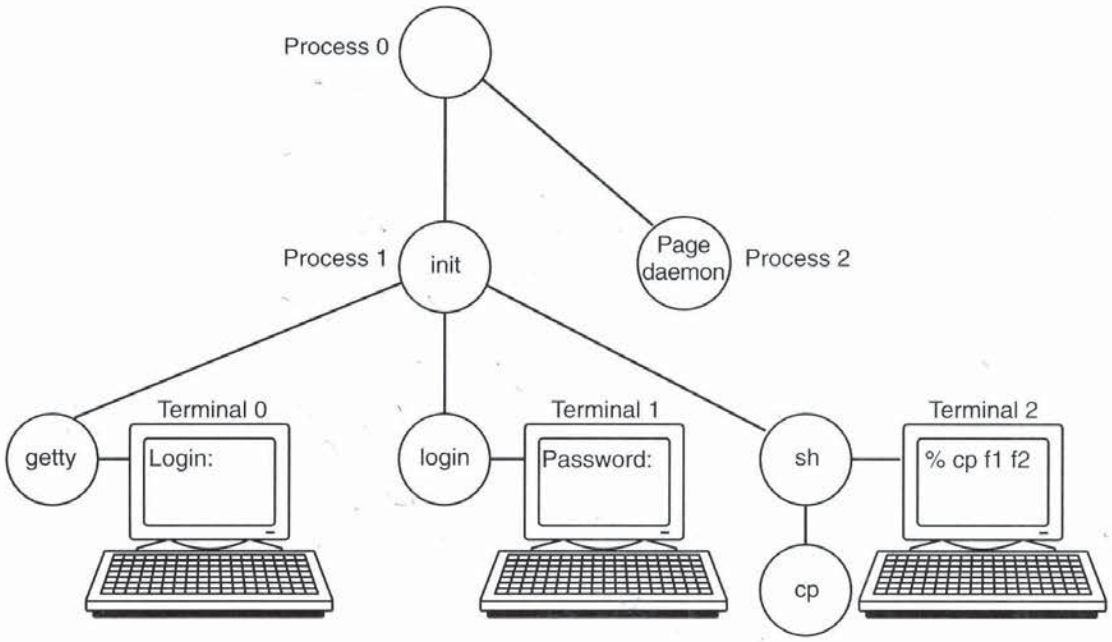


Figure 10-12. The sequence of processes used to boot some UNIX systems.

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

```
cp f1 f2
```

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

10.4 MEMORY MANAGEMENT IN UNIX

The UNIX memory model is straightforward, to make programs portable and to make it possible to implement UNIX on machines with widely differing memory management units, ranging from essentially nothing (e.g., the original IBM PC) to sophisticated paging hardware. This is an area of the design that has

barely changed in decades. It has worked well so it has not needed much revision. We will now examine the model and how it is implemented.

10.4.1 Fundamental Concepts

Every UNIX process has an address space consisting of three segments: text, data, and stack. An example process' address space is depicted in Fig. 10-13(a) as process *A*. The **text segment** contains the machine instructions that form the program's executable code. It is produced by the compiler and assembler by translating the C, C++, or other program into machine code. The text segment is normally read-only. Self modifying programs went out of style in about 1950 because they were too difficult to understand and debug. Thus the text segment neither grows nor shrinks nor changes in any other way.

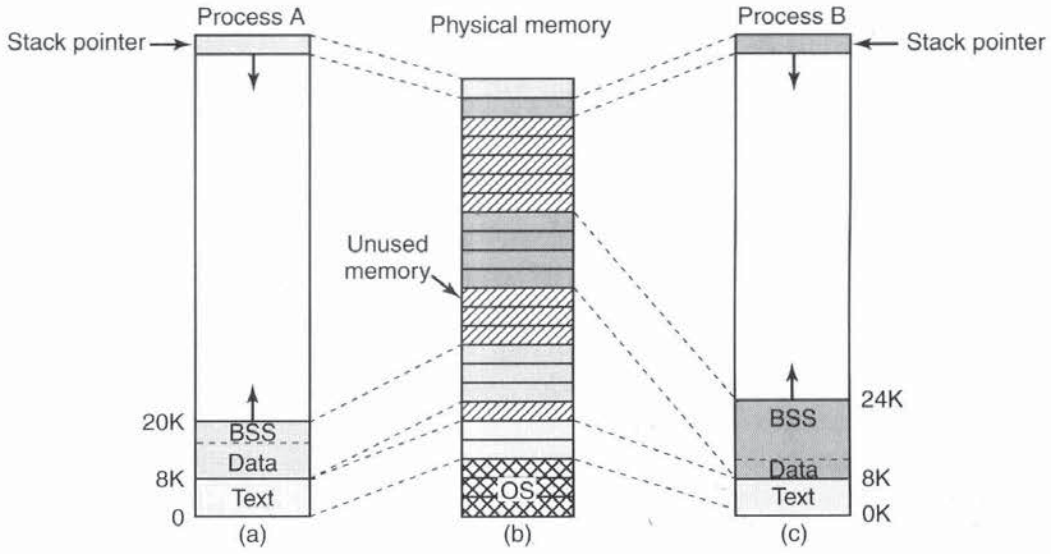


Figure 10-13. (a) Process *A*'s virtual address space. (b) Physical memory. (c) Process *B*'s virtual address space.

The **data segment** contains storage for the program's variables, strings, arrays, and other data. It has two parts, the initialized data and the uninitialized data. For historical reasons, the latter is known as the **BSS**. The initialized part of the data segment contains variables and compiler constants that need an initial value when the program is started.

For example, in C it is possible to declare a character string and initialize it at the same time. When the program starts up, it expects that the string has its initial value. To implement this construction, the compiler assigns the string a location in the address space, and ensures that when the program is started up, this location contains the proper string. From the operating system's point of view, initialized data are not all that different from program text—both contain bit patterns produced by the compiler that must be loaded into memory when the program starts.

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

However, to save space in the executable file, this is not done. Instead, the file contains all the explicitly initialized variables following the program text. The uninitialized variables are all gathered together after the initialized ones, so all the compiler has to do is put a word in the header telling how many bytes to allocate.

To make this point more explicit, consider Fig. 10-13(a) again. Here the program text is 8 KB and the initialized data is also 8 KB. The uninitialized data (BSS) is 4 KB. The executable file is only 16 KB (text + initialized data), plus a short header that tells the system to allocate another 4 KB after the initialized data and zero it before starting the program. This trick avoids storing 4 KB of zeros in the executable file.

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

The third segment is the stack segment. On most machines, it starts at or near the top of the virtual address space and grows down toward 0. If the stack grows below the bottom of the stack segment, a hardware fault normally occurs, and the operating system lowers the bottom of the stack segment by one page. Programs do not explicitly manage the size of the stack segment.

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

```
cp src dest
```

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

When two users are running the same program, such as the editor, it would be possible, but inefficient, to keep two copies of the editor's program text in memory at once. Instead, most UNIX systems support **shared text segments**. In Fig. 10-13(a) and Fig. 10-13(c) we see two processes, *A* and *B*, that have the same

text segment. In Fig. 10-13(b) we see a possible layout of physical memory, in which both processes share the same piece of text. The mapping is done by the virtual memory hardware.

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

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

Many versions of UNIX support **memory-mapped files**. This feature makes it possible to map a file onto a portion of a process' address space so the file can be read and written as if it were a byte array in memory. Mapping a file in makes random access to it much easier than using I/O system calls such as read and write. Shared libraries are accessed by mapping them in using this mechanism. In Fig. 10-14 we see a file that is mapped into two processes at the same time, at different virtual addresses.

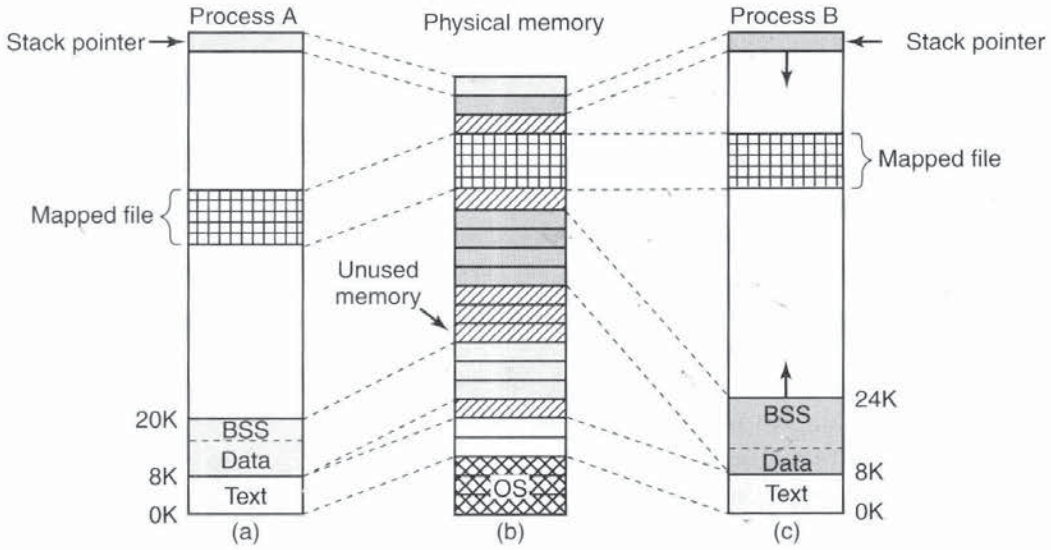


Figure 10-14. Two processes can share a mapped file.

An additional advantage of mapping a file in is that two or more processes can map in the same file at the same time. Writes to the file by any one of them are them instantly visible to the others. In fact, by mapping in a scratch file (which will be discarded after all the processes exit), this mechanism provides a

high-bandwidth way for multiple processes to share memory. In the most extreme case, two or more processes could map in a file that covers the entire address space, giving a form of sharing that is partway between separate processes and threads. Here the address space is shared (like threads), but each process maintains its own open files and signals, for example, which is not like threads. In practice, making two address spaces exactly correspond is never done, however.

10.4.2 Memory Management System Calls in UNIX

POSIX does not specify any system calls for memory management. This topic was considered too machine dependent for standardization. Instead, the problem was swept under the rug by saying that programs needing dynamic memory management can use the *malloc* library procedure (defined by the ANSI C standard). How *malloc* is implemented is thus moved outside the scope of the POSIX standard. In some circles, this approach is known as passing the buck.

In practice, most UNIX systems have system calls for managing memory. The most common ones are listed in Fig. 10-15. Brk specifies the size of the data segment by giving the address of the first byte beyond it. If the new value is greater than the old one, the data segment becomes larger; otherwise it shrinks.

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

Figure 10-15. Some system calls relating to memory management. The return code *s* is −1 if an error has occurred; *a* and *addr* are memory addresses, *len* is a length, *prot* controls protection, *flags* are miscellaneous bits, *fd* is a file descriptor, and *offset* is a file offset.

The mmap and munmap system calls control memory-mapped files. The first parameter to mmap, *addr*, determines the address at which the file (or portion thereof) is mapped. It must be a multiple of the page size. If this parameter is 0, the system determines the address itself and returns it in *a*. The second parameter, *len*, tells how many bytes to map. It, too, must be a multiple of the page size. The third parameter, *prot*, determines the protection for the mapped file. It can be marked readable, writable, executable, or some combination of these. The fourth parameter, *flags*, controls whether the file is private or sharable, and whether *addr* is a requirement or merely a hint. The fifth parameter, *fd*, is the file descriptor for the file to be mapped. Only open files can be mapped, so to map a file in, it must first be opened. Finally, *offset* tells where in the file to begin the mapping. It is not necessary to start the mapping at byte 0; any page boundary will do.

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

10.4.3 Implementation of Memory Management in UNIX

Prior to 3BSD, most UNIX systems were based on swapping, which worked as follows. When more processes existed than could be kept in memory, some of them were swapped out to disk. A swapped out process was always swapped out in its entirety (except possibly for shared text). A process was thus either in memory or on disk.

Swapping

Movement between memory and disk was handled by the upper level of the two-level scheduler, known as the **swapper**. Swapping from memory to disk was initiated when the kernel ran out of free memory on account of one of the following events:

1. A fork system call needed memory for a child process.
2. A brk system call needed to expand a data segment.
3. A stack became larger and ran out of the space allocated to it.

In addition, when it was time to bring in a process that had been on disk too long, it was frequently necessary to remove another process to make room for it.

To choose a victim to evict, the swapper first looked at the processes that were blocked waiting for something (e.g., terminal input). Better to remove a process that could not run than one that could. If one or more were found, the one whose priority plus residence time was the highest was chosen. Thus a process that had consumed a large amount of CPU time recently was a good candidate, as was one that had been in memory for a long time, even if it was mostly doing I/O. If no blocked process was available, then a ready process was chosen based on the same criteria.

Every few seconds, the swapper examined the list of processes currently swapped out to see if any of them were ready to run. If any were, the one that had been on disk longest was selected. Next, the swapper checked to see if this was going to be an easy swap or a hard one. An easy swap was one for which enough free memory currently existed, so that no process had to be removed to make room for the new one. A hard swap required removing one or more processes. An easy swap was implemented by just bringing in the process. A hard one was implemented by first freeing up enough memory by swapping out one or more processes, then bringing in the desired process.

This algorithm was then repeated until one of two conditions was met: (1) no processes on disk were ready to run, or (2) memory was so full of processes that

had just been brought in that there was no room left for any more. To prevent thrashing, no process was ever swapped out until it had been in memory for 2 sec.

Free storage in memory and on the swap device was kept track of by linked lists of holes. When storage was needed on either one, the appropriate hole list was read using the first fit algorithm, which returned the first sufficiently large hole it could find. The hole was then reduced in size, leaving only the residual part not needed. list.

Paging in UNIX

All versions of UNIX for the PDP-11 and Interdata machines, as well as the initial VAX implementation, were based on swapping, as just described. Starting with 3BSD, however, Berkeley added paging in order to handle the ever-larger programs that were being written. Virtually all UNIX systems now have demand paging, tracing their origins to 3BSD. Below we will describe the 4BSD design, but the System V one is closely based on 4BSD and is almost the same.

The basic idea behind paging in 4BSD is simple: a process need not be entirely in memory in order to run. All that is actually required is the user structure and the page tables. If these are swapped in, the process is deemed "in memory" and can be scheduled to run. The pages of the text, data, and stack segments are brought in dynamically, one at a time, as they are referenced. If the user structure and page table are not in memory, the process cannot be run until the swapper brings them in.

Berkeley UNIX does not use the working set model or any other form of prepaging because doing so requires knowing which pages are in use and which are not. Because the VAX did not have page reference bits, this information was not easily available (although it can be obtained in software at the cost of substantial additional overhead).

Paging is implemented partly by the kernel and partly by a new process called the **page daemon**. The page daemon is process 2 (process 0 is the swapper, and process 1 is init, as shown in Fig. 10-12). Like all daemons, the page daemon is started up periodically so it can look around to see if there is any work for it to do. If it discovers that the number of pages on the list of free memory pages is too low, it initiates action to free up more pages.

Main memory in 4BSD is organized as shown in Fig. 10-16. It consists of three parts. The first two parts, the kernel and core map, are pinned in memory (i.e., never paged out). The rest of memory is divided into page frames, each of which can contain a text, data, or stack page, a page table page, or be on the free list.

The **core map** contains information about the contents of the page frames. Core map entry 0 describes page frame 0, core map entry 1 describes page frame 1, and so forth. With 1-KB page frames and 16-byte core map entries, less than 2 percent of the memory is taken up by the core map. The first two items in the

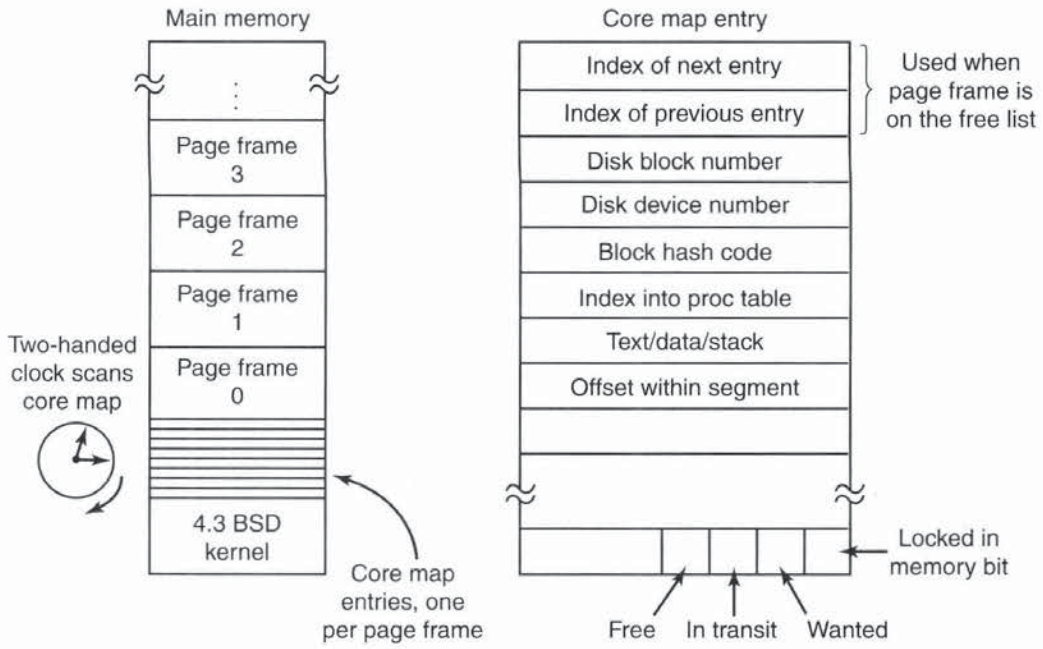


Figure 10-16. The core map in 4BSD.

core map entry shown in Fig. 10-16 are used only when the corresponding page frame is on the free list. Then they are used to hold a doubly linked list stringing together all the free page frames. The next three entries are used when the page frame contains information. Each page in memory also has a location on some disk where it is put when it is paged out. These entries are used to find the disk location where the page is stored. The next three entries give the process table entry for the page's process, which segment it is in, and where it is located in that segment. The last one shows some of the flags needed by the paging algorithm.

When a process is started, it may get a page fault because one or more of its pages are not present in memory. If a page fault occurs, the operating system takes the first page frame on the free list, removes it from the list, and reads the needed page into it. If the free list is empty, the process is suspended until the page daemon has freed a page frame.

The Page Replacement Algorithm

The page replacement algorithm is executed by the page daemon. Every 250 msec it is awakened to see if the number of free page frames is at least equal to a system parameter called *lotsfree* (typically set to 1/4 of memory). If insufficient page frames are free, the page daemon starts transferring pages from memory to disk until *lotsfree* page frames are available. If the page daemon discovers that more than *lotsfree* page frames are on the free list, it knows that it has nothing to

do so it just goes back to sleep. If the machine has plenty of memory and few active processes, the page daemon sleep nearly all the time.

The page daemon uses a modified version of the clock algorithm. It is a global algorithm, meaning that when removing a page, it does not take into account whose page is being removed. Thus the number of pages each process has assigned to it varies in time.

The basic clock algorithm works by scanning the page frames circularly (as though they lay around the circumference of a clock). On the first pass, when the hand points to a page frame, the usage bit is cleared. On the second pass, any page frame that has not been accessed since the first pass will have its usage bit still cleared, and will be put on the free list (after writing it to disk, if it is dirty). A page frame on the free list retains its contents, which can be recovered if that page is needed before it is overwritten.

On a machine like the VAX that had no usage bits, when the clock hand pointed to a page frame on the first pass, the software usage bit was cleared and the page marked as invalid in the page table. When the page was next accessed, a page fault occurred, which allowed the operating system to set the software usage bit. The effect was the same as having a hardware usage bit, but the implementation was much more complicated and slower. Thus the software paid the price for poor hardware design.

Originally, Berkeley UNIX used the basic clock algorithm, but it was discovered that with large memories, the passes took too long. The algorithm was then modified to the **two-handed clock algorithm**, symbolized at the left of Fig. 10-16. With this algorithm, the page daemon maintains two pointers into the core map. When it runs, it first clears the usage bit at the front end, and then checks the usage bit at the back hand, after which it advances both hands. If the two hands are kept close together, then only very heavily used pages have much of a chance of being accessed between the time the front hand passes by and the time the back one does. If the two hands are 359 degrees apart (meaning the back hand is just ahead of the front hand), we are back to the original clock algorithm. Each time the page daemon runs, the hands rotate less than a full revolution, the amount depending on how far they have to go to get the number of pages on the free list up to *lotsfree*.

If the system notices that the paging rate is too high and the number of free pages is always way below *lotsfree*, the swapper is started to remove one or more processes from memory so that they will no longer compete for page frames. The 4BSD swap out algorithm is as follows. First the swapper looks to see if any processes have been idle for 20 sec or more. If any exist, the one that has been idle the longest is swapped out. If none exist, the four largest processes are examined and the one that has been in memory the longest is swapped out. If need be, the algorithm is repeated until enough memory has been recovered.

Every few seconds the swapper checks to see if any ready processes on the disk should be brought in. Each process on the disk is assigned a value that is a

function of how long it has been swapped out, its size, the value it set using nice (if any), and how long it was sleeping before being swapped out. The function is weighted to usually bring in the process that has been out the longest, unless it is extremely large. The theory is that bringing in large processes is expensive, so they should not be moved too often. Swap-in only occurs if there are enough free pages, so that when the inevitable page faults start occurring, there will be page frames for them. Only the user structure and the page tables are actually brought in by the swapper. The text, data, and stack pages are paged in as they are used.

Each segment of each active process has a place on disk where it resides when it is paged or swapped out. Data and stack segments go to a scratch device, but program text is paged in from the executable binary file itself. No scratch copy is used for program text.

Paging in System V is fundamentally similar to that in 4BSD, which is not entirely surprising since the Berkeley version had been stable and running for years before paging was added to System V. Nevertheless, there are two interesting differences.

First, System V uses the original one-handed clock algorithm, instead of the two-handed one. Furthermore, instead of putting an unused page on the free list on the second pass, a page is only put there if it is unused for n consecutive passes. While this decision does not free pages as quickly as the Berkeley algorithm, it greatly increases the chance that a page once freed will not be needed again quickly.

Second, instead of a single variable *lotsfree*, System V has two variables, *min* and *max*. Whenever the number of free page frames falls below *min*, the page daemon is started to free up more pages. The daemon continues to run until there are *max* free page frames. This approach eliminates a potential instability in 4BSD. Consider a situation in which the number of free page frames is one less than *lotsfree*, so the page daemon runs to free one page and bring it up to *lotsfree*. Then another page fault occurs, using up one page frame, and reducing the number of available frames below *lotsfree* again, so the daemon has to run again. By setting *max* substantially above *min*, whenever the page daemon runs, it builds up a sufficient inventory so it does not have to run again for a substantial time.

Memory Management in Linux

Each Linux process on a 32-bit machine gets 3 GB of virtual address space for itself, with the remaining 1 GB reserved for its page tables and other kernel data. The kernel's 1 GB is not visible when running in user mode, but becomes accessible when the process traps into the kernel. The address space is created when the process is created and is overwritten on an exec system call.

The virtual address space is divided into homogeneous, contiguous, page-aligned areas or regions. That is to say, each area consists of a run of consecutive pages with the same protection and paging properties. The text segment and

mapped files are examples of areas (see Fig. 10-14). There can be holes in the virtual address space between the areas. Any memory reference to a hole results in a fatal page fault. The page size is fixed, for example, 4 KB for the Pentium and 8 KB for the Alpha.

Each area is described in the kernel by a *vm_area_struct* entry. All the *vm_area_structs* for a process are linked together in a list sorted on virtual address so all the pages can be found. When the list gets too long (more than 32 entries), a tree is created to speed up searching. The *vm_area_struct* entry lists the area's properties. These include the protection mode (e.g., read only or read/write), whether it is pinned in memory (not pageable), and which direction it grows in (up for data segments, down for stacks).

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

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

Linux uses a three-level paging scheme. Although this scheme was put into the system for the Alpha, it is also used (in degenerate form) for all architectures. Each virtual address is broken up into four fields, as shown in Fig. 10-17. The directory field is used as an index into the global directory, of which there is a private one for each process. The value found is a pointer to one of the page middle tables, which is again indexed by a field from the virtual address. The selected entry points to the final page table, which is indexed by the page field of the virtual address. The entry found here points to the page needed. On the Pentium, which uses two-level paging, each page middle directory has only one entry, so the global directory entry effectively chooses the page table to use.

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

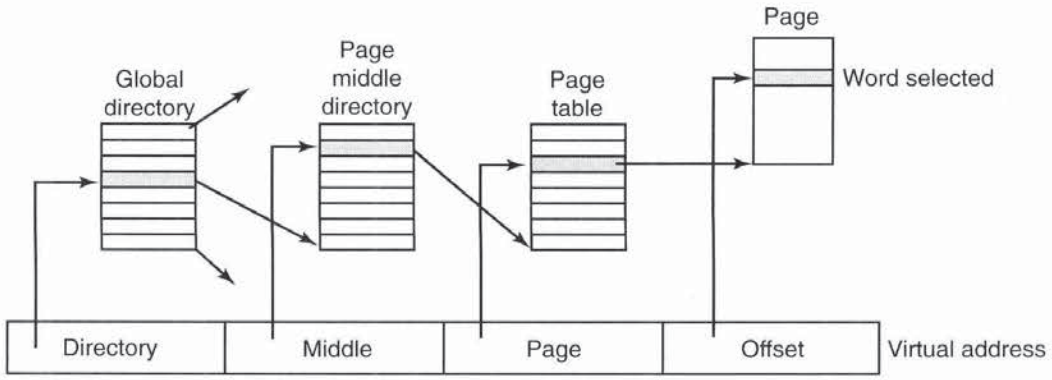


Figure 10-17. Linux uses three-level page tables.

In addition, Linux supports dynamically loaded modules, generally device drivers. These can be of arbitrary size and each one must be allocated a contiguous piece of kernel memory. As a consequence of these requirements, Linux manages physical memory in such a way that it can acquire an arbitrary-sized piece of memory at will. The algorithm it uses is known as the **buddy algorithm** and is described below.

The basic idea for managing a chunk of memory is as follows. Initially memory consists of a single contiguous piece, 64 pages in the simple example of Fig. 10-18(a). When a request for memory comes in, it is first rounded up to a power of two, say 8 pages. The full memory chunk is then divided in half, as shown in (b). Since each of these pieces is still too large, the lower piece is divided in half again (c) and again (d). Now we have a chunk of the correct size, so it is allocated to the caller, as shown shaded in (d).

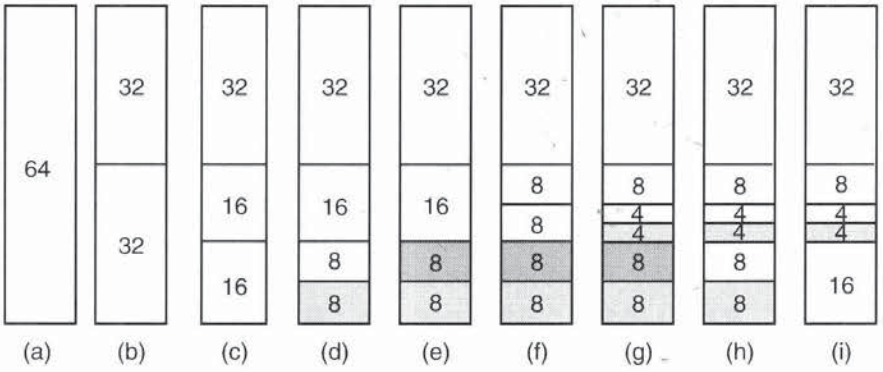


Figure 10-18. Operation of the buddy algorithm.

Now suppose that a second request comes in for 8 pages. This can be satisfied directly now (e). At this point a third request comes in for 4 pages. The smallest available chunk is split (f) and half of it is claimed (g). Next, the second of the 8-page chunks is released (h). Finally, the other 8-page chunk is released.

Since the two adjacent 8-page chunks that were just freed are buddies, that is, originated from the same 16-page chunk, they are merged to get the 16-page chunk back (i).

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

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

To alleviate this problem, Linux has a second memory allocation that takes chunks using the buddy algorithm but then carves slabs (smaller units) from them and manages the smaller units separately. A third memory allocator is also available when the requested memory need only be contiguous in virtual space, but not in physical memory. All these memory allocators are derived from those in System V.

Linux is a demand-paged system with no prepaging and no working set concept (although there is a system call in which a user can give a hint that a certain page may be needed soon, in the hopes it will be there when needed). Text segments and mapped files are paged to their respective files on disk. Everything else is paged to either the paging partition, if present, or one of up to 8 fixed-length paging files. Paging files can be added and removed dynamically and each one has a priority. Paging to a separate partition, accessed as a raw device, is more efficient than paging to a file for several reasons. First, the mapping between file blocks and disk blocks is not needed (saves disk I/O reading indirect blocks). Second, the physical writes can be of any size, not just the file block size. Third, a page is always written contiguously to disk; with a paging file, it may or may not be.

Pages are not allocated on the paging device or partition until they are needed. Each device and file starts with a bitmap telling which pages are free. When a page without backing store has to be tossed out of memory, the highest priority paging partition or file that still has space is chosen and a page allocated on it. Normally, the paging partition, if present, has higher priority than any paging file. The disk location is written into the page table.

Page replacement works as follows. Linux tries to keep some pages free so they can be claimed as needed. Of course, this pool must be continually replenished, so the real paging algorithm is how this happens. At boot time, *init* starts up a page daemon, *kswapd*, that runs once a second. It checks to see if there are enough free pages available. If so, it goes back to sleep for another second, although it can be awakened early if more pages are suddenly needed. The page daemon's code consists of a loop that it executes up to six times, with increasingly high urgency. Why six? Probably the author of the code thought four was not enough and eight was too many. Linux is sometimes like that.

The body of the loop makes calls to three procedures, each of which tries to reclaim different kinds of pages. The urgency value is passed as a parameter telling the procedure how much effort to expend to reclaim some pages. Usually, this means how many pages to inspect before giving up. The effect of this algorithm is to first take the easy pages from each category before going after the hard ones. When enough pages have been reclaimed, the page daemon goes back to sleep.

The first procedure tries to reclaim pages from the paging cache and the file system buffer cache that have not been referenced recently, using a clock-like algorithm. The second procedure looks for shared pages that none of the users seems to be using much. The third procedure, which tries to reclaim ordinary user pages, is the most interesting, so let us take a quick look at it.

First, a loop is made over all the processes to see which one has the most pages currently in memory. Once that process is located, all of its *vm_area_structs* are scanned and all the pages are inspected in virtual address order starting from where we left off last time. If a page is invalid, absent from memory, shared, locked in memory, or being used for DMA, it is skipped. If the page has its reference bit on, the bit is turned off and the page is spared. If the reference bit is turned off, out goes the page, so this part of the algorithm is similar to clock (except that the pages are not scanned in FIFO order).

If the page is chosen for eviction and it is clean, it is abandoned immediately. If it is dirty and has a backing store page on disk, it is scheduled for a disk write. Finally, if it is dirty and has no backing store page, it goes into the paging cache, from which it might conceivably get a reprieve later if it is reused before it is actually paged out. The idea of scanning the pages in virtual address order is based on the hope that pages near each other in virtual address space will tend to be used or not used as a group, so they should be written to disk as a group and later brought in together as a group.

One other aspect of the memory management system that we have not yet mentioned is a second daemon, *bdflush*. It wakes up periodically (and in some cases is explicitly awakened) to check if too large a fraction of the pages are dirty. If so, it starts writing them back to disk.

10.5 INPUT/OUTPUT IN UNIX

The I/O system in UNIX is fairly straightforward. Basically, all I/O devices are made to look like files and are accessed as such with the same read and write system calls that are used to access all ordinary files. In some cases, device parameters must be set, and this is done using a special system call. We will study these issues in the following sections.

10.5.1 Fundamental Concepts

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

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

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

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

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

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

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

Although most character special files cannot be randomly accessed, they often need to be controlled in ways that block special files do not. Consider, for example, input typed on the keyboard and displayed on the screen. When a user makes a typing error and wants to erase the last character typed, he presses some key. Some people prefer to use backspace, and others prefer DEL. Similarly, to erase the entire line just typed, many conventions abound. Traditionally @ was used, but with the spread of email (which uses @ within email address), many systems

have adopted CTRL-U or some other character. Likewise, to interrupt the running program, some special key must be hit. Here too, different people have different preferences.

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

Networking

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

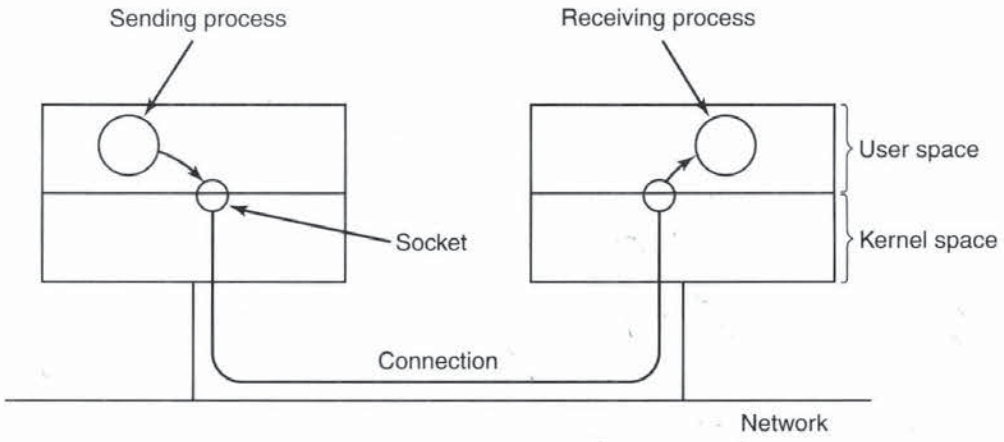


Figure 10-19. The uses of sockets for networking.

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

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

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

The first socket type allows two processes on different machines to establish the

equivalent of a pipe between them. Bytes are pumped in at one end and they come out in the same order at the other. The system guarantees that all bytes that are sent arrive and in the same order they were sent.

The second type is 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 are 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 domain 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 one 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.

10.5.2 Input/Output System Calls in UNIX

Each I/O device in a UNIX 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 the terminal. Whether each one is a separate system call or they share a single system call or something else is implementation dependent.

The first four 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. This asymmetry still persists, with some telephone companies offering inbound service at 1.5 Mbps and outbound service at 384 Kbps, 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 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 still exists on most UNIX systems.

10.5.3 Implementation of Input/Output in UNIX

I/O in UNIX 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 structure arrays, *bdevsw* for block special files or *cdevsw* for character special files. 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 UNIX means adding a new entry to one of these tables and supplying the corresponding procedures to handle the various operations on the device.

The most important fields in the *cdevsw* array for a typical system might look as 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 *cdevsw* array to select the proper row (structure), then calls the function in the corresponding structure member (column) to have the work performed. Thus each row contains pointers to functions contained in one 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 fields of a typical *cdevsw* table.

Each driver is split into two parts. The top half runs in the context of the caller and interfaces to the rest of UNIX. 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 UNIX is covered in detail in (Egan and Teixeira, 1992).

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 that part of the system that does I/O on block special files (e.g., disks) is to minimize the number of actual transfers that must be done. To accomplish this goal, UNIX systems have a **buffer cache** between the disk drivers and the file system, as illustrated in Fig. 10-22. The buffer 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 any purpose (i-node, directory, or data), a check is first made to see if it is in the buffer cache. If so, it is taken from there and a disk access is avoided. The buffer cache greatly improves system performance.

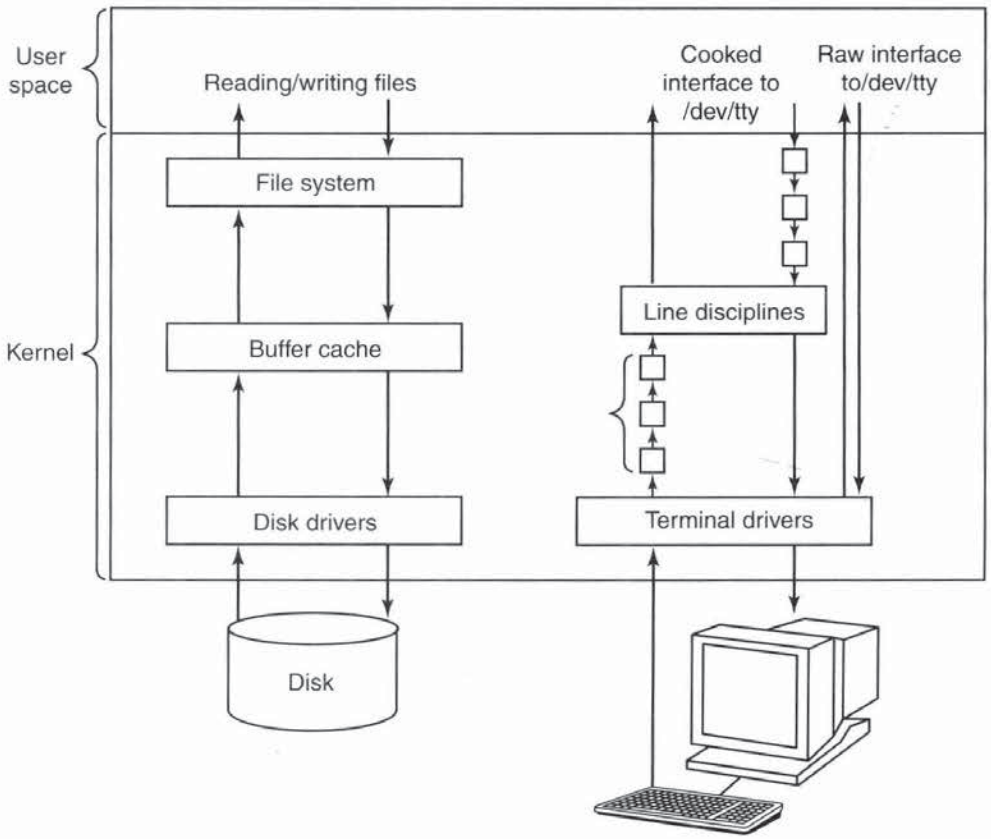


Figure 10-22. The UNIX I/O system in BSD.

If the block is not in the buffer cache, it is read from the disk into the buffer cache and from there, copied to where it is needed. Since the buffer cache has room for only a fixed number of blocks, some algorithm is needed to manage it. Usually, the blocks in the cache are linked together in a linked list. Whenever a block is accessed, it is moved to the head of the list. When a block must be removed from the cache to make room for a new block, the one at the rear of the chain is selected, since it is the least recently used block.

The buffer cache works for writes as well as for reads. When a program writes a block, it goes to the cache, not to the disk. Only when the cache fills up and the buffer must be reclaimed is the block forced out onto the disk. To avoid having blocks stay too long in the cache before being written to the disk, all the dirty blocks are written to the disk every 30 seconds.

For decades, UNIX device drivers have been statically linked into the kernel so they were all present in memory when the system was booted every time. Given the environment in which UNIX grew up, mostly 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 it 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 *cdevsw* or *bdevsw*, 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. Some modern UNIX systems also support loadable modules now, too.

10.5.4 Streams

Since character special files deal with character streams and do not move blocks of information between memory and disk, they do not use the buffer cache. Instead, in the first versions of UNIX, each character device driver did all the work it needed for its device. However, in the course of time it became clear that many drivers were duplicating code present in other drivers such as code for buffering, flow control, and network protocols. Two different solutions were developed for structuring and modularizing the character drivers. We will now briefly examine each of them in turn.

The BSD solutions builds on data structures present in classical UNIX systems called **C-lists**, shown as small boxes in Fig. 10-22. Each one is a block of up to 64 characters, plus a count and a pointer to the next block. As characters arrive from terminals and other character devices, they are buffered in a chain of these blocks.

When a user process reads from */dev/tty* (e.g., standard input), the characters are not passed directly from the C-list to the process. Instead, they pass through a piece of kernel code called a **line discipline**. The line discipline acts like a filter, accepting the raw character stream from the terminal driver, processing it, and

producing what is called a **cooked character stream**. In the cooked stream, local line editing has been done (i.e., erased characters and lines have been removed), carriage returns have been mapped onto line feeds, and other special processing has been completed. The cooked stream is passed to the process. However, if the process wants to interact on every character, it can put the line in raw mode, in which case the line discipline will be bypassed.

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 System V solution, **streams**, devised by Dennis Ritchie, is more general and is shown in Fig. 10-23. (System V also has a buffer cache for block special files, but since that is essentially the same as in BSD it is not shown here.) The basic idea of a stream is to be able to connect a user process to a driver and be able to insert processing modules into the stream dynamically, during execution. To some extent, a stream is the kernel analog of pipelines in user space.

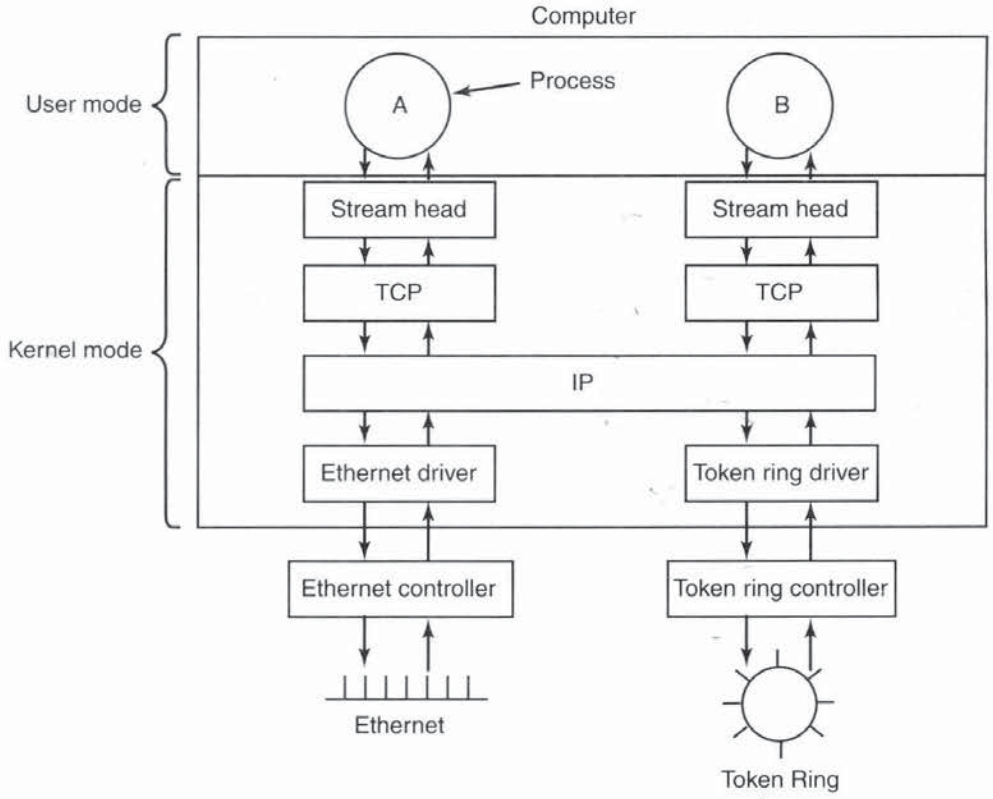


Figure 10-23. An example of streams in System V.

A stream always has a stream head on the top and connects to a driver on the bottom. As many modules as needed can be inserted into the stream. Processing can be bidirectional, so each module may need a section for reading (from the driver) and one for writing (to the driver). When a user process writes to a stream, code in the stream head interprets the system call and packs the data into stream buffers that are passed from module to module downward, with each module performing whatever transformations it is supposed to. Each module maintains a read queue and a write queue, so buffers are processed in the correct order. Modules have well-defined interfaces, defined by the streams infrastructure, so unrelated modules can be plugged together.

The example of Fig. 10-23 shows how streams are used when using the Internet TCP protocol over two different kinds of local area networks, Ethernet and Token ring. In particular, it illustrates another important feature of streams—multiplexing. A multiplexing module can take one stream and split it into multiple streams or can take multiple streams and merge them into one stream. The IP module in this configuration does both.

10.6 THE UNIX FILE SYSTEM

The most visible part of any operating system, including UNIX, is the file system. In the following sections we will examine the basic ideas behind the UNIX 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. The UNIX 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, UNIX nevertheless provides a powerful and elegant file system.

10.6.1 Fundamental Concepts

A UNIX 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 were originally restricted to 14 arbitrary characters, but Berkeley UNIX increased the limit to 255 characters, and this has been adopted by System V and most other versions as well. 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.f90* is typically a FORTRAN 90 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.Z*, which is probably a 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 usually 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. 10-24.

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

Figure 10-24. Some important directories found in most UNIX systems.

There are two ways to specify file names in UNIX, both to the shell and when opening a file from within a program. The first way is using 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/mos2/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 *mos2* which contains the file *chap-10*.

Absolute path names are often long and inconvenient. For this reason, UNIX 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/mos2* is the working directory, then the shell command

```
cp chap-10 backup1
```

has exactly the same effect as the longer command

```
cp /usr/ast/books/mos2/chap-10 /usr/ast/books/mos2/backup1
```

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. UNIX provides a solution to this problem 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. 10-25(a). Fred and Lisa are working together on a project, and each one needs frequent 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. 10-25(b), after which he can use *x* to mean */usr/lisa/x*.

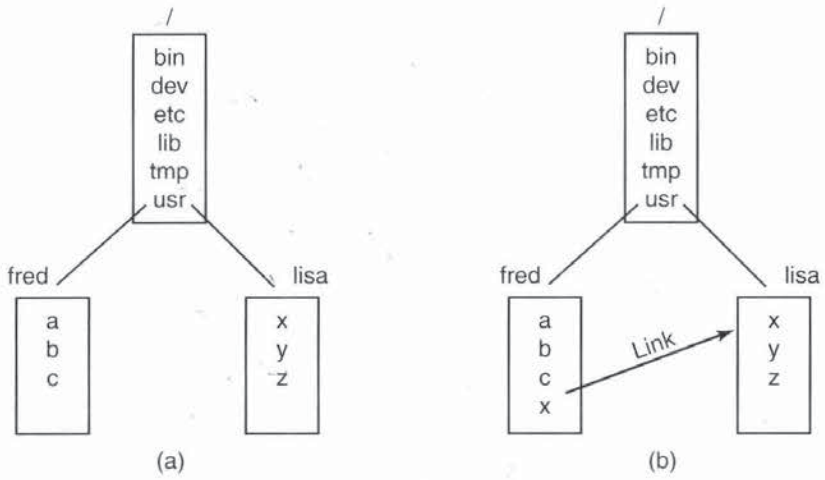


Figure 10-25. (a) Before linking. (b) After linking.

In the example just discussed, we suggested that before linking, the only way for Fred to refer to Lisa's file *x* was 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, UNIX 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 normally have

at least two disks—a hard disk and a diskette 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 depicted in Fig. 10-26(a). Here we have a hard disk, which we will call *C*:, and a diskette, which we will call *A*:. 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 example, to copy the file *x* to the directory *d*, (assuming *C:* is the default), one would type

```
cp  A:/x  /a/d/x
```

This is the approach taken by systems like MS-DOS, Windows 98, and VMS.

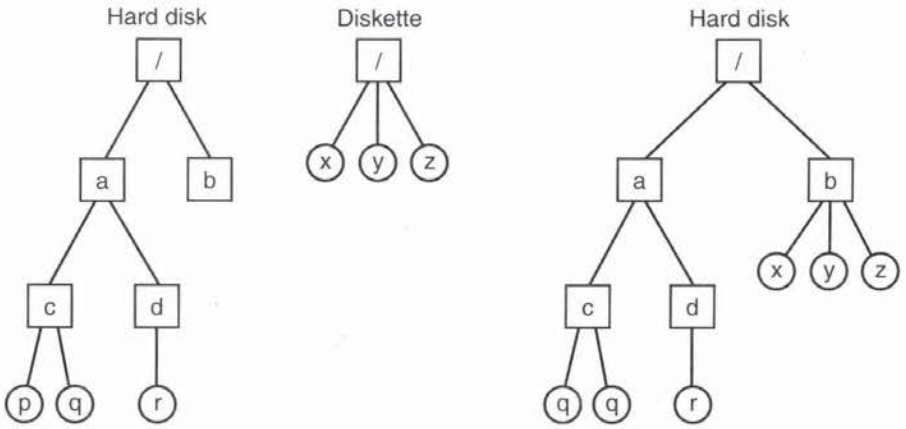


Figure 10-26. (a) Separate file systems. (b) After mounting.

The UNIX solution is to allow one disk to be mounted in another disk's file tree. In our example, we could mount the diskette on the directory */b*, yielding the file system of Fig. 10-26(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 UNIX file system is **locking**. In some applications, two or more processes may be using the same file at the same time, which may lead to race conditions. One solution is to program the application with critical regions. However, if the processes belong to independent users who do not even know each other, this kind of coordination is generally inconvenient.

Consider, for example, a database consisting of many files in one or more directories that are accessed by unrelated users. It is certainly possible to associate a

semaphore with each directory or file and achieve mutual exclusion by having processes do a down operation on the appropriate semaphore before accessing the data. The disadvantage, however, is that a whole directory or file is then made inaccessible, even though only one record may be needed.

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

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

When placing a lock, a process must specify whether it wants to block or not in the event that the lock cannot be placed. If it chooses to block, when the existing lock has been removed, the process is unblocked and the lock is placed. If the process chooses not to block when it cannot place a lock, the system call returns immediately, with the status code telling whether the lock succeeded or not.

Locked regions may overlap. In Fig. 10-27(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. 10-27(b). Finally, *C* locks bytes 2 through 11. As long as all these locks are shared, they can co-exist. Now consider what happens if a process tries to acquire an exclusive lock to byte 9 of the file of Fig. 10-27(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 UNIX

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.

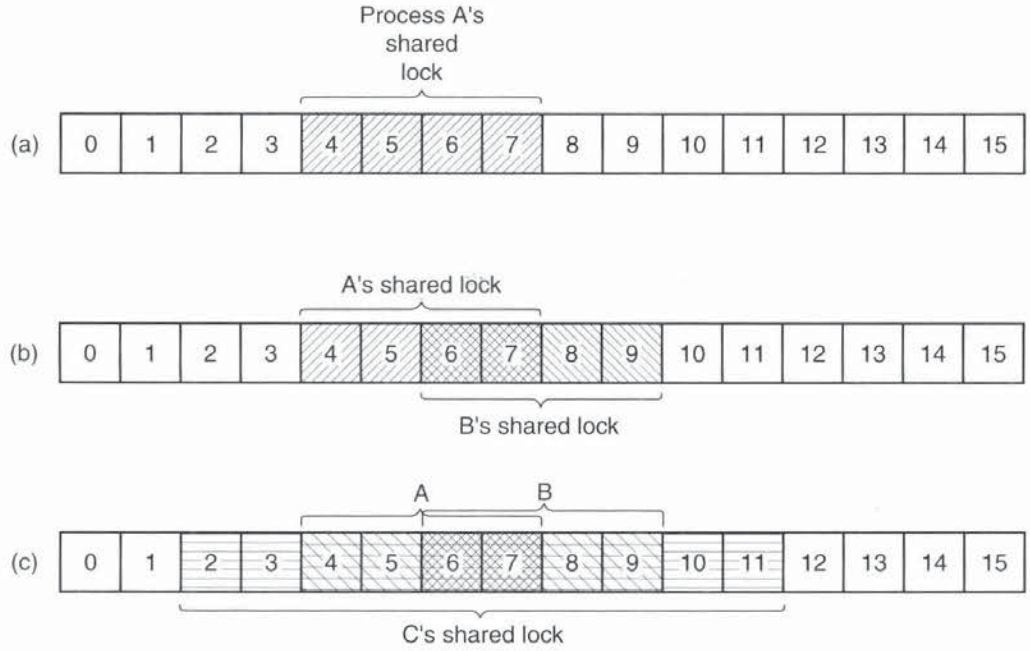


Figure 10-27. (a) A file with one lock. (b) Addition of a second lock. (c) A third lock.

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 as its result 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.

Now let us continue looking at the principal file system calls, which are listed in Fig. 10-28. To read or write an existing file, the file must first be opened using open. 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.

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

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 10-28. 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.

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 most 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 (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 one is a file position; the third one 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 was changed. Slightly ironically, lseek is the only file system call that can never cause an actual disk seek because all it does is update the current file position, which is a number in memory.

For each file, UNIX 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 one is a pointer to a structure where the information requested is to be

put. The fields in the structure are shown in Fig. 10-29. 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 10-29. 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 where a simple concept (redirection) with a simple implementation (file descriptors 0 and 1) leads to a powerful tool (connecting programs in arbitrary ways without having to modify them at all).

The last system call in Fig. 10-28 is fcntl. It is used to lock and unlock files 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. 10-30. Directories are created and destroyed using mkdir and rmdir, respectively. A directory can only be removed if it is empty.

As we saw in Fig. 10-25, 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

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 10-30. 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.

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. 10-30 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 UNIX File System

In this section we will describe the implementation of the traditional UNIX file system. Afterward, we will discuss the Berkeley improvements. Other file systems are also in use. All UNIX systems can handle multiple disk partitions, each with a different file system on it.

A classical UNIX, disk partition contains a file system with the layout illustrated in Fig. 10-31. Block 0 is not used by UNIX and often contains code to boot the computer. Block 1 is the **superblock**. It contains critical 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). Destruction of the superblock will render the file system unreadable.

Following the superblock are the **i-nodes** (short for index-nodes, but never called that, although some lazy people drop the hyphen and call them **inodes**).

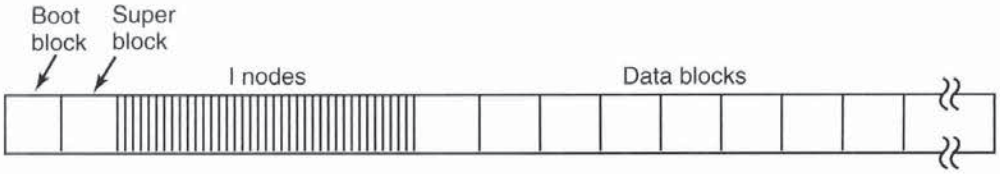


Figure 10-31. Disk layout in classical UNIX systems.

They are numbered from 1 up to some maximum. Each i-node is 64 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. It is this scatter that the Berkeley improvements were designed to reduce.

A directory in the traditional (i.e., V7) file system consists of an unsorted collection of 16-byte entries. Each entry contains a file name (up to 14 arbitrary characters), and the number of the file's i-node, as shown in Fig. 6-37. To open a file in the working directory, the system just reads the directory, comparing the name to be looked up to each entry until it either finds the name or concludes that it is not present.

If the file is present, the system extracts the i-node number, and uses this 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 varies somewhat from UNIX version to UNIX version. As a bare minimum, all the fields returned by the stat system call must be present to make stat work (see Fig. 10-29). In Fig. 10-32 we show the format used by all AT&T versions from Version 7 through System V.

Looking up an absolute path name such as */usr/ast/file* is slightly more complicated. First, the system locates the root directory, which generally uses i-node 2, especially when i-node 1 is reserved for bad block handling. Then it looks up the string "usr" in the root directory, to get the i-node number of the */usr* directory. 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.

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	39	Address of first 10 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 10-32. Structure of the i-node in System V.

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 about 20).

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 UNIX. 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 line

```
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. 10-33. The trick is to introduce a new table, the **open file description** table between the file descriptor table and the i-node table, and put the file position (and read/write bit) there. In this figure, the parent is the shell and the child is first *p1* and later *p2*. When the shell forks off *p1*, its user structure (including the file descriptor table) is an exact copy of the shell's, so both of them point to the same open file description table entry. When *p1* finishes, the shell's file descriptor is still pointing to the open file description containing *p1*'s file position. When the shell now forks off *p2*, the new child automatically inherits the file position, without either it or the shell even having to know what that position is.

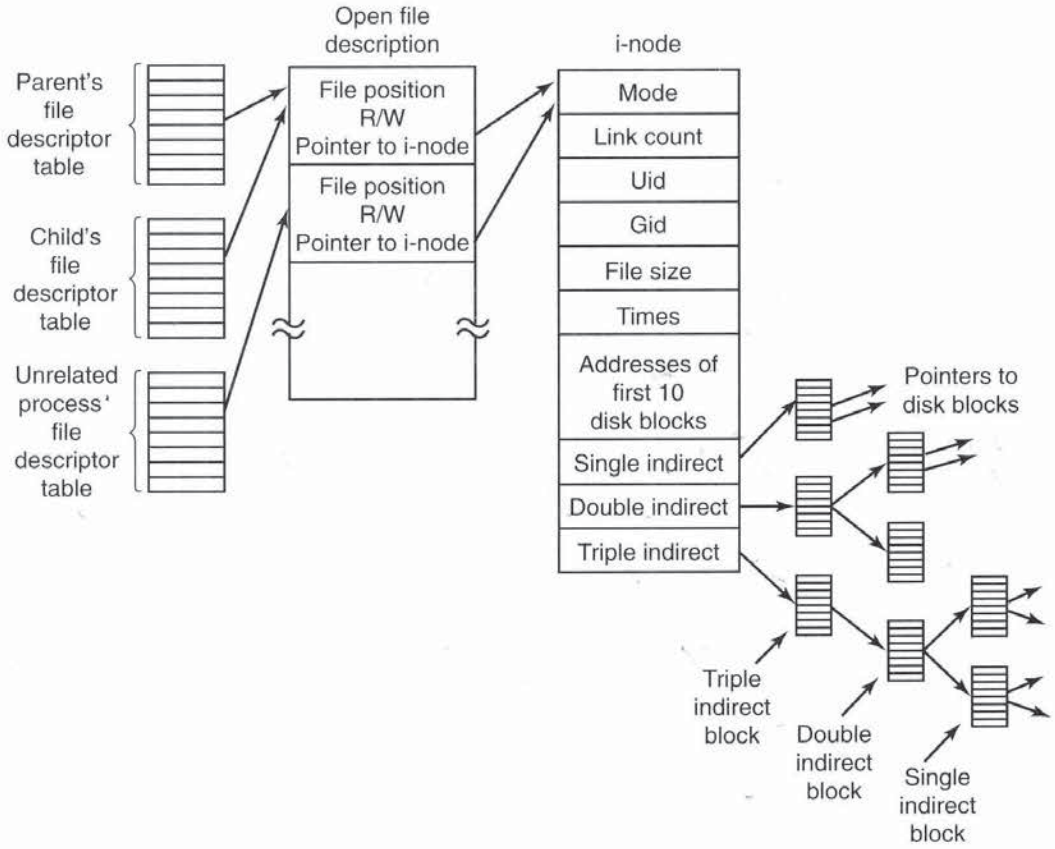


Figure 10-33. The relation between the file descriptor table, the open file description table, and the i-node table.

However, if an unrelated process opens the file, it gets its own open file description entry, 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 10 blocks of the file. If the file position falls in the first 10 blocks, the block is read and the data are copied to the user. For files longer than 10 blocks, a field in the i-node contains the disk address of a **single indirect block**, as shown in Fig. 10-33. 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 266 KB in total.

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

The Berkeley Fast File System

The description above explains how the classical UNIX file system works. Let us now take a look at the improvements Berkeley made to it. First, directories were reorganized. Instead of limiting file names to 14 characters, the limit was set to 255 characters. Of course, changing the structure of all the directories meant that programs that naively read directories (which was and is still legal) and which expected a sequence of 16-byte entries, no longer worked. To provide portability across the two kinds of directories, Berkeley provided the system calls opendir, closedir, readdir, and rewinddir to allow programs to read directories without having to know their internal structure. Long file names and these system calls for reading directories were later added to all other versions of UNIX and to POSIX.

The BSD directory structure allows file names up to 255 characters and is illustrated in Fig. 10-34. 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. 10-34 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 telling how big the entry is (in bytes), possibly including some padding after the name. This field is needed to find the next entry for the case that the file name is

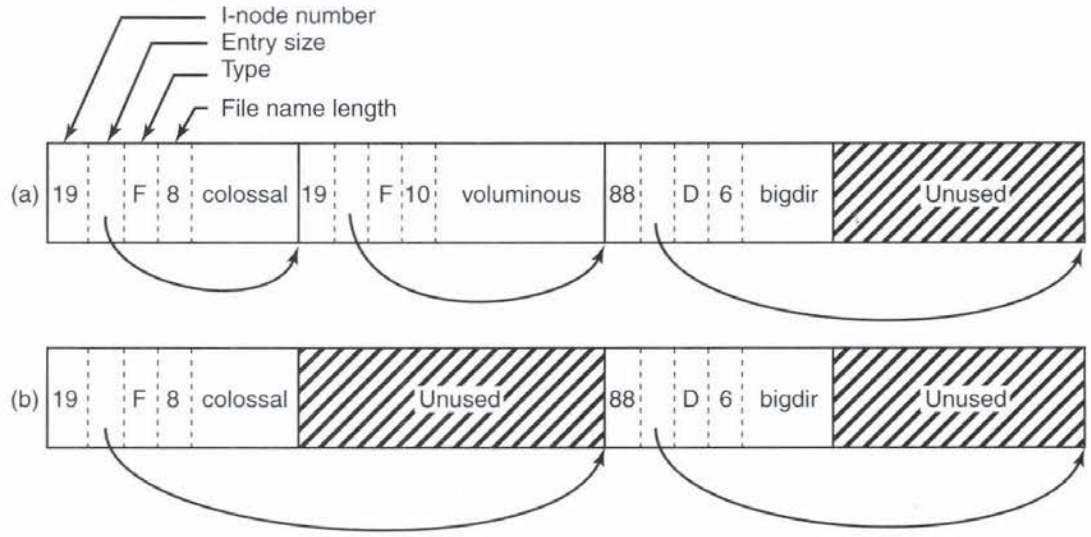


Figure 10-34. (a) A BSD directory with three files. (b) The same directory after the file *voluminous* has been removed.

padded by an unknown length. That is the meaning of the arrow in Fig. 10-34. Then comes the type field: file, directory, etc. 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. 10-34(b) we see the same directory after the entry for *voluminous* has been removed. All that is 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. To improve search performance, name caching was added to BSD. Before looking up a name, the system first checks the cache. If a cache hit occurs, the linear search can be bypassed.

The second major change Berkeley made to the file system was the division of the disk up into **cylinder groups**, each with its own superblock, i-nodes, and data blocks. The idea behind this change is to keep the i-node and data blocks of a file close together, to avoid long seeks. Whenever possible, blocks are allocated in the cylinder group containing the i-node.

The third change was the introduction of two block sizes instead of just one. For storing large files, it is more efficient to have a small number of large blocks rather than many small ones. On the other hand, many UNIX files are small, so having only large blocks would be wasteful of disk space. Having two sizes allows efficient transfers for large files and space efficiency for small ones. The price paid is considerable extra complexity in the code.

The Linux File System

The initial Linux file system was the MINIX file system. However, due to the fact that it limited file names to 14 characters (in order to be compatible with UNIX Version 7) and its maximum file size was 64 MB, there was interest in better file systems almost from the beginning. 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 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 that has become the main file system. However, Linux still supports over a dozen file systems using the NFS file system (described in the next section). When Linux is linked, a choice is offered of which file systems should be built into the kernel. Other ones can be dynamically loaded as modules during execution, if need be.

Ext2 is very similar to the Berkeley Fast File system with some small differences. The Ext2 disk layout is shown in Fig. 10-35. Instead of using cylinder groups, which mean almost nothing with today's virtual disk geometries, it divides the disk into groups of blocks without regard to where the cylinder boundaries fall. Each block group starts with a superblock, which tells how many blocks and i-nodes there are, gives the block size, etc. 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.

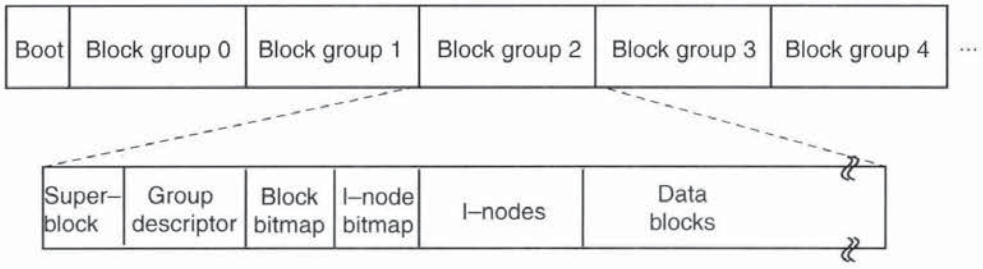


Figure 10-35. Layout of the Linux Ext2 file system.

Two bitmaps keep track of the free blocks and free i-nodes, respectively. 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 the latter is not in practice. Next come the i-nodes themselves. Each i-node is 128 bytes, twice the size of the standard UNIX i-nodes of Fig. 10-32. The extra space is used as follows. Instead of 10 direct block addresses and 3 indirect block addresses, Linux allows 12 direct and 3 indirect addresses. Furthermore, the addresses have been extended from 3 bytes to 4 bytes, to handle disk partitions larger than 2^{24} blocks (16 GB), which was already a problem in UNIX. In addition, fields are

reserved for pointers to access control lists for a finer degree of protection, but these are still on the drawing board. The rest of the i-node is reserved for future use. History shows that unused bits do not remain that way for long.

Operation of the file system is similar to the Berkeley fast file system. One difference with Berkeley, however, is the use of standard 1 KB throughout. The Berkeley fast file systems uses 8-KB blocks and then fragments them into 1-KB pieces if need be. Ext2 just does it the easy way. Like the Berkeley system, when a file grows, Ext2 tries to put the block in the same block group as the rest of the file, preferably right after the previous last block. Also, when a new file is added to a directory, Ext2 tries to put it into the same block group as its directory. New directories are scattered uniformly across the disk.

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 UNIX 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 later). In this section we will examine Sun Microsystem's **NFS** (**Network File System**), which is used on all modern UNIX systems (and some non-UNIX systems) to join the file systems on separate computers into one logical whole. Three aspects of NFS are of interest: the architecture, the protocol, and the implementation. We will now examine these in turn.

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 in fact, 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. 10-36.

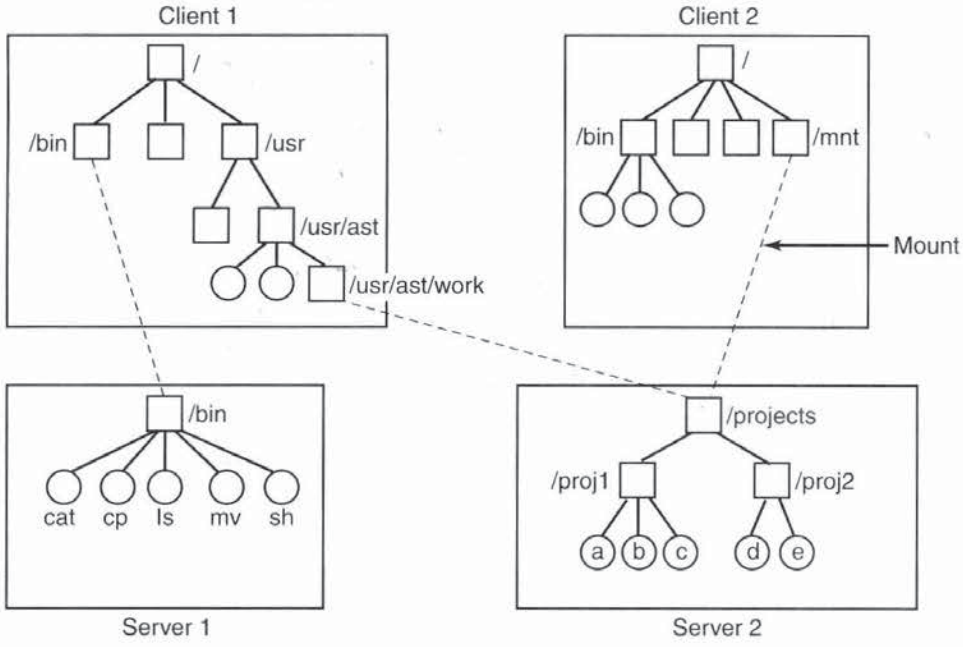


Figure 10-36. Examples of remote mounted file systems. Directories are shown as squares and files are shown as circles.

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 it possible for anyone to be 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 UNIX 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 UNIX 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. Also, they

can also access file attributes, such as file mode, size, and time of last modification. Most UNIX system calls are supported by NFS, with the perhaps surprising exception 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 UNIX file semantics. For example, in UNIX 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?—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 enabled, 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 UNIX systems use a three-layer implementation similar to that of Fig. 10-37. 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, analogous to the table of i-nodes for open files in UNIX. In ordinary UNIX, an i-node is indicated uniquely by a (device, i-node number) pair. Instead, the VFS layer has an entry, called a **v-node** (**virtual i-node**), for every open file. V-

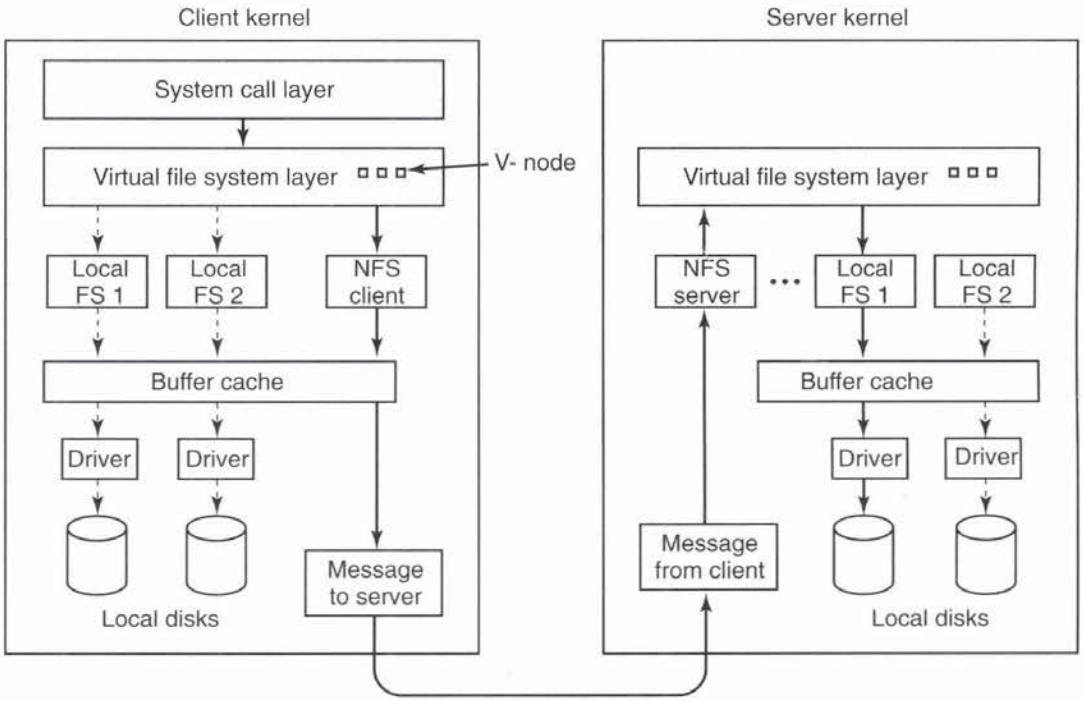


Figure 10-37. The NFS layer structure.

nodes are used to tell whether the file is local or remote. For remote files, enough information is provided to be able to access them. For local files, the file system and i-node are recorded because modern UNIX systems can support multiple file systems (e.g., V7, Berkeley FFS, ext2fs, /proc, FAT, etc.). Although VFS was invented to support NFS, most modern UNIX 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. 10-37 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. 10-37). 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 bytes 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 that 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.

10.7 SECURITY IN UNIX

Despite the name, 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 UNIX.

10.7.1 Fundamental Concepts

The user community for a UNIX 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 **GID**s **Group ID**s). 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. Originally, a user could be in only one group, but some versions of UNIX now allow a user to be in more than one group at the same time. For simplicity, we will not discuss this feature further.

The basic security mechanism in UNIX 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. 10-38.

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 10-38. Some example file protection modes.

The first two entries in Fig. 10-38 are clear, allowing 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 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 UNIX 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 only change 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 UNIX

There are only a small number of system calls relating to security. The most important ones are listed in Fig. 10-39. 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 10-39. 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 only allowed for the superuser. They change a file's owner, and a process' UID and GID.

10.7.3 Implementation of Security in UNIX

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 *sh*, 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.

Security in Linux is essentially the same as in UNIX. All the UNIX security features are implemented and there is little else in this area.

10.8 SUMMARY

UNIX began life as a minicomputer timesharing system, but is now used on machines ranging from notebook computers to supercomputers. Three interfaces to it exist: the shell, the C library, and the system calls themselves. 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 lean and mean, roughly 100 calls, each of which doing what is needed and no more.

The key concepts in UNIX 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 UNIX uses two key data structures, the process table and the user structure. The former is always in memory, but the latter can be swapped or paged out. Process creation is done by duplicating the process table entry, and then the memory image. Scheduling is done using a priority-based algorithm that favors interactive processes.

The memory model consists of three segments per process: text, data, and stack. Memory management used to be done by swapping, but is now done by paging in most UNIX systems. The core map keeps track of the state of each page, and the page daemon uses a clock algorithm to keep enough free pages around.

I/O devices are accessed using special files, each of which has a major device number and a minor device number. Block device I/O uses a buffer cache to reduce the number of disk accesses. An LRU algorithm is used to manage the cache. Character I/O can be done in raw or cooked mode. Line disciplines or streams are used to add features to character I/O.

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.

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

PROBLEMS

1. When the kernel catches system call, how does it know which system call it is supposed to carry out?
2. A directory contains the following files:

aardvark	feret	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*?
```

3. What does the following UNIX shell pipeline do?

```
grep nd xyz | wc –l
```

4. Write a UNIX pipeline that prints the eighth line of file *z* on standard output.
5. Why does UNIX distinguish between standard output and standard error, when both default to the terminal?
6. 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?

7. When the UNIX 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?

8. About how long does it take to fork off a child process under the following conditions: text size = 100 KB, data size = 20 KB, stack size = 10 KB, process table size = 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.

9. As megabyte programs became more common, the time spent executing the fork system call grew proportionally. Worse yet, nearly all of this time was wasted, since most programs call exec shortly after forking. To improve performance, Berkeley invented a new system call, vfork, in which the child shares the parent's address space, instead of getting its own copy of it. Describe a situation in which a poorly-behaved child can do something that makes the semantics of vfork fundamentally different from those of fork.

10. If a UNIX process has a *CPU_usage* value of 20, how many ΔT intervals does it take to decay to 0 if it does not get scheduled for a while? Use the simple decay algorithm given in the text.

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

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

13. Why do you think the designers of UNIX made it impossible for a process to send a signal to a another process that is not in its process group?

14. 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?

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

16. 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.

17. In every process' entry in the process table, the PID of the process' parent is stored. Why?

18. 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?

19. The Linux scheduler computes the goodness for real-time processes by adding the priority to 1000. Could a different constant have been chosen and still have the algorithm make the same choices as the regular one?

20. When booting UNIX (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 the extra step needed? Surely it would be simpler to have the bootstrap loader in sector 0 just load the operating system directly.

21. 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?

22. In 4BSD, each core map entry contains an index for the next entry on the free list, which is used when the current entry is on the free list. This field is 16 bits. Pages are 1 KB. Do these sizes have any implications for the total amount of memory BSD can support? Explain your answer.

23. In BSD, 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?

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

25. 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?

26. 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?

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

28. 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 co-exist without being merged into one block? If so, give an example of how this can happen. If not, show that it is impossible.

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

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

31. 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.

32. Consider the locked file of Fig. 10-27(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 kind of problems are introduced into the semantics by this situation. Propose and defend two solutions.

33. Suppose that an lseek system call seeks to a negative offset in a file. Given two possible ways of dealing with it.

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

35. Some tape drives have numbered blocks and the ability to overwrite a particular block in place without disturbing the blocks in front or behind it. Could such a device hold a mounted UNIX file system?

36. In Fig. 10-25, both Fred and Lisa have access to the file *x* in their respective directories after linking. Is this access completely symmetrical in the sense anything one of them can do with it the other one can too?

37. 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.

38. 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.

39. A UNIX i-node has 10 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?

40. 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?

41. Why does LRU work for managing the buffer cache, whereas it rarely works for keeping track of pages in a virtual memory system?

42. UNIX has a system call, sync, that flushes the buffer cache back to disk. At boot time, a program called *update* is started. Once every 30 sec it calls sync, then goes to sleep for 30 sec. Why does this program exist?

43. After a system crash and reboot, a recovery program is usually run. Suppose that 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?

44. Make an educated guess as to which UNIX system call is the fastest.

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

46. Based on the information presented in this chapter, if a Linux ext2 file system were to be put on a 1.44 Mbyte 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.

47. 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?

48. A professor shares files with his students by placing them in a publically accessible directory on the CS department's UNIX system. One day he realizes that a file placed there the previous day was left world-writable. He changes the permissions and verifies the file is identical to his master copy. The next day he finds the file has been changed. How could this have happened and how could it have been prevented?

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

50. 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.

51. Write a dumb terminal program to connect two UNIX or Linux workstations via the serial ports. Use the POSIX terminal management calls to configure the ports.

11

CASE STUDY 2: WINDOWS 2000

Windows 2000 is a modern operating system that runs on high-end desktop PCs and servers. In this chapter we will examine various aspects of it, starting with a brief history, then moving on to its architecture. After this we will look at processes, memory management, I/O, the file system, and finally, security. One thing we will not look at is networking as that could easily fill a chapter (or an entire book) all by itself.

11.1 HISTORY OF WINDOWS 2000

Microsoft operating systems for desktop and laptop PCs can be divided into three families: MS-DOS, Consumer Windows (Windows 95/98/Me), and Windows NT. Below we will briefly sketch each of these families.

11.1.1 MS-DOS

In 1981, IBM, at the time the biggest and most powerful computer company in the world, produced the 8088-based IBM PC. The PC came equipped with a 16-bit real-mode, single-user, command-line oriented operating system called MS-DOS 1.0. The operating system was provided by Microsoft, a tiny startup, mostly known at that time for its BASIC interpreter used on 8080 and Z-80 systems. This operating system consisted of 8 KB of memory resident code and was

closely modeled on CP/M, a tiny operating system for the 8-bit 8080 and Z80 CPUs. Two years later, a much more powerful 24-KB operating system, MS-DOS 2.0, was released. It contained a command line processor (shell), with a number of features borrowed from UNIX.

When Intel came out with the 286 chip, IBM built a new computer around it, the PC/AT, released in 1986. AT stood for Advanced Technology, because the 286 ran at a then-impressive 8 MHz and could address—with great difficulty—all of 16 MB of RAM. In practice, most systems had at most 1 MB or 2 MB, due to the great expense of so much memory. The PC/AT came equipped with Microsoft's MS-DOS 3.0, by now 36 KB. Over the years, MS-DOS continued to acquire new features, but it was still a command-line oriented system.

11.1.2 Windows 95/98/Me

Inspired by the user interface of the Apple Lisa, the forerunner to the Apple Macintosh, Microsoft decided to give MS-DOS a graphical user interface (shell) that it called **Windows**. Windows 1.0, released in 1985, was something of a dud. Windows 2.0, designed for the PC-AT and released in 1987, was not much better. Finally, Windows 3.0 for the 386 (released in 1990), and especially its successors 3.1 and 3.11, caught on and were huge commercial successes. None of these early versions of Windows were true operating systems, but more like graphical user interfaces 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 grinding halt.

The release of Windows 95 in August 1995 still did not completely eliminate MS-DOS, although it transferred nearly all the features from the MS-DOS part to the Windows part. Together, Windows 95 and the new MS-DOS 7.0 contained most of the features of a full-blown operating system, including virtual memory, process management, and multiprogramming. However, Windows 95 was not a full 32-bit program. It contained large chunks of old 16-bit assembly code (as well as some 32-bit code) and still used the MS-DOS file system, with nearly all its limitations. The only major change to the file system was the addition of long file names in place of the 8 + 3 character file names allowed in MS-DOS.

Even with the release of Windows 98 in June 1998, MS-DOS was still there (now called version 7.1) and running 16-bit code. Although yet more functionality migrated from the MS-DOS part to the Windows part, and a disk layout suitable for larger disks was now standard, under the hood, Windows 98 was not very different from Windows 95. The main difference was the user interface, which integrated the desktop and the Internet more closely. It was precisely this integration that attracted the attention of the U.S. Dept. of Justice, which then sued Microsoft claiming that it was an illegal monopoly, an accusation Microsoft vigorously denied. In April 2000, a U.S. Federal court agreed with the government.

In addition to containing a large lump of old 16-bit assembly code in the kernel, Windows 98 had two other serious problems. First, although it was a multiprogramming system, the kernel itself was not reentrant. If a process was busy manipulating some kernel data structure and then suddenly its quantum ran out and another process started running, the new process might find the data structure in an inconsistent state. To prevent this type of problem, after entering the kernel, most processes first acquired a giant mutex covering the whole system before doing anything. While this approach eliminated potential inconsistencies, it also eliminated much of the value of multiprogramming since processes were frequently forced to wait for unrelated processes to leave the kernel before they could enter it.

Second, each Windows 98 process had a 4-GB virtual address space. Of this, 2 GB was completely private to the process. However, the next 1 GB was shared (writably) among all other processes in the system. The bottom 1 MB was also shared among all processes to allow all of them to access the MS-DOS interrupt vectors. This sharing facility was heavily used by most Windows 98 applications. As a consequence, a bug in one program could wipe out key data structures used by unrelated processes, leading to them all crashing. Worse yet, the last 1 GB was shared (writably) with the kernel and contained some critical kernel data structures. Any rogue program that overwrote these data structures with garbage could bring down the system. The obvious solution of not putting kernel data structures in user space was not possible because this feature was essential to making old MS-DOS programs work under Windows 98.

In the millennium year, 2000, Microsoft brought out a minor revision to Windows 98 called **Windows Me** (**Windows Millennium Edition**). Although it fixed a few bugs and added a few features, under the covers it is essentially Windows 98. The new features included better ways to catalog and share images, music, and movies, more support for home networking and multiuser games, and more Internet-related features, such as support for instant messaging and broadband connections (cable modems and ADSL). One interesting new feature was the ability to restore the computer to its previous settings after a misconfiguration. If a user reconfigures the system (e.g., changing the screen from 640×480 to 1024×768) and it no longer works, this feature makes it possible to revert to the last known working configuration.

11.1.3 Windows NT

By the late 1980s, Microsoft realized that building a modern 32-bit operating system on top of the leaky 16-bit MS-DOS probably was not the best way to go. It recruited David Cutler, one of the key designers of DEC's VMS operating system, to work for Microsoft and gave him the job of leading a team to produce a brand-new 32-bit Windows compatible operating system from the ground up. This new

system, later called **Windows NT (Windows New Technology)**, was intended for mission-critical business applications as well as for home users. At the time, mainframes still ruled the (business) world, so designing an operating system on the assumption that companies would use personal computers for anything important was a visionary goal, but one that history has shown to be a very good one. Features such as security and high reliability, clearly lacking on the MS-DOS-based versions of Windows, were high on the agenda for (Windows) NT. Cutler's background with VMS clearly shows in various places, with there being more than a passing similarity between the design of NT and that of VMS.

The project succeeded and the first version, called Windows NT 3.1, was released in 1993. This initial release number was chosen to match the number of Microsoft's then-popular 16-bit Windows 3.1 system. Microsoft expected that NT would rapidly replace Windows 3.1 because it was technically a far superior system.

Much to its surprise, nearly all users preferred to stick with the old 16-bit system they knew, rather than upgrade to an unknown 32-bit system they did not know, however better it might be. Furthermore, NT required far more memory than Windows 3.1 and there were no 32-bit programs for it to run, so why bother? The failure of NT 3.1 to catch on in the marketplace was the reason Microsoft decided to build a 32-bit-ish version of Windows 3.1, namely Windows 95. The continued user resistance to NT then caused Microsoft to produce Windows 98 and finally Windows Me; each one claimed to be the very last release of the MS-DOS-based systems.

Despite the fact that nearly all consumers and most businesses ignored NT 3.1 for desktop systems, it did acquire a small following in the server market. A few new 3.x releases with small changes occurred in 1994 and 1995. These slowly began to acquire more following among desktop users as well.

The first major upgrade to NT came with NT 4.0 in 1996. This system had the power, security, and reliability of the new operating system, but also sported the same user interface as the by-then very popular Windows 95. This compatibility made it much easier for users to migrate from Windows 95 to NT, and many of them did so. Some of the differences between Windows 95/98 and Windows NT are summarized in Fig. 11-1.

From the beginning, NT was designed to be portable, so it was written almost entirely in C, with only a tiny bit of assembly code for low-level functions such as interrupt handling. The initial release consisted of 3.1 million lines of C for the operating system, libraries, and the environment subsystems (discussed below). When NT 4.0 came out, the code base had grown to 16 million lines of code, still mostly C, but with a small amount of C++ in the user interface part. By this time the system was highly portable, with versions running on the Pentium, Alpha, MIPS, and PowerPC, among other CPUs. Some of these have been dropped since then. The story of how NT was developed is given in the book *Showstopper* (Zachary, 1994). The book also tells a lot about the key people involved.

Item	Windows 95/98	Windows NT
Full 32-bit system?	No	Yes
Security?	No	Yes
Protected file mappings?	No	Yes
Private addr space for each MS-DOS prog?	No	Yes
Unicode?	No	Yes
Runs on	Intel 80x86	80x86, Alpha, MIPS, ...
Multiprocessor support?	No	Yes
Re-entrant code inside OS?	No	Yes
Plug and play?	Yes	No
Power management?	Yes	No
FAT-32 file system?	Yes	Optional
NTFS file system	No	Yes
Win32 API?	Yes	Yes
Run all old MS-DOS programs?	Yes	No
Some critical OS data writable by user?	Yes	No

Figure 11-1. Some differences between Windows 98 and Windows NT.

11.1.4 Windows 2000

The release of NT following NT 4.0 was originally going to be called NT 5.0. However, in 1999, Microsoft changed the name to Windows 2000, mostly in an attempt to have a neutral name that both Windows 98 users and NT users could see as a logical next step for them. To the extent that this approach succeeds, Microsoft will have a single main operating system built on reliable 32-bit technology but using the popular Windows 98 user interface.

Since Windows 2000 really is NT 5.0, it inherits many properties from NT 4.0. It is a true 32-bit (soon to be 64-bit) multiprogramming system with individually protected processes. Each process has a private 32-bit (soon 64-bit) demand-paged virtual address space. The operating system runs in kernel mode, whereas user processes run in user mode, providing complete protection (with none of the protection flaws of Windows 98). Processes can have one or more threads, which are visible to, and scheduled by, the operating system. It has Dept. of Defense C2 security for all files, directories, processes, and other shareable objects (at least, if the floppy disk is removed and the network is unplugged). Finally, it also has full support for running on symmetric multiprocessors with up to 32 CPUs.

The fact that Windows 2000 really is NT 5.0 is visible in many places. For example, the system directory is called *\winnt* and the operating system binary (in *\winnt\system32*) is called *ntoskrnl.exe*. Right clicking on this file to examine its

properties shows that its version number is *5.xxx.yyy.zzz*, where the *5* stands for NT 5, *xxx* is the release number, *yyy* is the build (compilation) number, and *zzz* the minor variant. Also, many of the files in *\winnt* and its subdirectories have *nt* in their names, such as *ntvdm*, NT's virtual MS-DOS emulator.

Windows 2000 is more than just a better NT 4.0 with the Windows 98 user interface. To start with, it contains a number of other features previously found only in Windows 98. These include complete support for plug-and-play devices, the USB bus, IEEE 1394 (FireWire), IrDA (the infrared link between portable computers and printers), and power management, among others. In addition, a number of new features not present in any other Microsoft operating system have been added, including active directory directory service, security using Kerberos, support for smart cards, system monitoring tools, better integration of laptop computers with desktop computers, a system management infrastructure, and job objects. Also, the main file system, NTFS, has been extended to support encrypted files, quotas, linked files, mounted volumes, and content indexing, for example. Another novel NTFS feature is the single instance store, which is a kind of copy-on-write link in which two users can share a linked file until one of them writes on it, at which time a copy is made automatically.

One other major improvement is internationalization. NT 4.0 came in separate versions for different languages with the text strings embedded in the code. Installing an English software package on a Dutch computer often caused parts of the operating system to stop using Dutch and start using English because certain files containing code and text strings were overwritten. This problem has been eliminated. Windows 2000 has a single binary that runs everywhere in the world. An installation, or even an individual user, can choose the language to use at run time because all the menu items, dialog strings, error reports, and other text strings have been removed from the operating system and put in separate directories, one per installed language. Like all previous versions of NT, Windows 2000 uses Unicode throughout the system to support languages not using the Latin alphabet, such as Russian, Greek, Hebrew, and Japanese.

One thing that Windows 2000 does not have is MS-DOS. It is simply not there in any form (nor was it there in NT). There is a command line interface, but this is a new 32-bit program that includes the old MS-DOS functionality and considerably new functionality as well.

Despite many portability features with regard to the code, hardware, language, etc., in one respect Windows 2000 is less portable than NT 4.0: it runs on only two platforms, the Pentium and the Intel IA-64. Originally NT supported additional platforms, including the PowerPC, MIPS, and Alpha, but over the years, Microsoft dropped one after another for commercial reasons.

Like previous versions of NT, Windows 2000 comes in several product levels, this time: Professional, Server, Advanced server, and Datacenter server. The differences between all these versions are minor however, with the same executable binary used for all versions. When the system is installed, the product type is

recorded in an internal database (the registry). At boot time, the operating system checks the registry to see which version it is. The differences are shown in Fig. 11-2.

Version	Max RAM	CPUs	Max clients	Cluster size	Optimized for
Professional	4 GB	2	10	0	Response time
Server	4 GB	4	Unlimited	0	Throughput
Advanced server	8 GB	8	Unlimited	2	Throughput
Datacenter server	64 GB	32	Unlimited	4	Throughput

Figure 11-2. The different versions of Windows 2000.

As can be seen from the figure, the differences include the maximum memory supported, the maximum number of CPUs (for a multiprocessor configuration), and the maximum number of clients that can be served. The cluster size relates to the ability of Windows 2000 to make two or four machines look like a single server to the outside world, a useful feature for Web servers, for example. Finally, the default parameters are tuned differently on Professional, to favor interactive programs over batch work, although these can easily be changed if desired. One last difference is that some extra software is provided on the servers and some extra tools are provided on Datacenter server for managing large jobs.

The reason for having multiple versions is simply marketing: this allows Microsoft to charge big companies more than they charge individuals for what is essentially the same product. This idea is not new, however, and hardly unique to Microsoft. For years, airlines having been charging business passengers much more, not only for Business Class, but also for Cattle Class if they want the luxury of buying the ticket a day before the flight instead of a month before the flight.

Technically, the way the version differences are maintained is that in a few places in the code, two variables are read from the registry, *ProductType* and *ProductSuite*. Depending on their values, slightly different code is executed. Changing these variables is in violation of the license. In addition, the system traps any attempt to change them and records the attempt at tampering in an indelible way so it can be detected later.

In addition to the basic operating system, Microsoft has also developed several tool kits for advanced users. These include the Support Tools, the Software Development Kit, the Driver Development Kit, and the Resource Kit. These include a large number of utilities and tools for tweaking and monitoring the system. The support tools are on the Windows 2000 CD-ROM in the directory *\support\tools*. The standard installation procedure does not install them, but there is a file *setup.exe* in that directory that does. The SDK and DDK are available to developers at *msdn.microsoft.com*. The Resource Kit is a Microsoft product in a box. There are also various third-party tools available for snooping on the Windows 2000 internals, including a nice set available for free at the Web site

www.sysinternals.com. Some of these even provide more information than the corresponding Microsoft tools.

Windows 2000 is an immensely complex system, now consisting of over 29 million lines of C code. If printed 50 lines per page and 1000 pages per bound book, the full code would occupy 580 volumes. This opus would occupy 23 running meters of shelf space (for the paperback version). If arranged in bookcases 1 m wide with 6 shelves per bookcase, the set would occupy a wall 4 m wide.

Just for fun, a comparison of a few operating system source code sizes is given in Fig. 11-3. However, this table should be taken with a grain (or better yet, a metric ton) of salt because what constitutes the operating system is different for different systems. For example, the entire window system and GUI is part of the kernel in Windows, but not in any UNIX version. It is simply a user process there. Counting X Windows adds another 1.5 million lines of code to all the UNIX versions, and that does not even count the GUI code (Motif, GNOME, etc.), which is also not part of the operating system in the UNIX world. Additionally, some systems include code for multiple architectures (e.g., five for 4.4 BSD and nine for Linux), with each architecture adding 10,000 to 50,000 lines of code. The reason Free BSD 1.0 has only 235,000 lines of code whereas 4BSD Lite, from which it is derived, has 743,000 lines is that support for all the obsolete architectures (e.g., the VAX) was dropped in Free BSD.

Also, the number of file systems, devices drivers, and libraries supplied varies greatly from system to system. In addition, Windows contains large amounts of test code that UNIX does not contain as well as some utilities and support for numerous languages besides English. Finally, the measurements were made by different people, which introduces considerable variance (e.g., did makefiles, headers, configuration files and documentation count and how much was there?). This is not like comparing apples with oranges; it is like comparing apples with telephones. However, all the counts within a single family came from the same source, so intrafamily counts are somewhat meaningful.

Despite all these disclaimers, two conclusions are fairly clear:

1. System bloat seems to be as inevitable as death and taxes.
2. Windows is much bigger than UNIX

Whether small is beautiful or big is beautiful is a matter of heated controversy. The argument for the former is that small size and a lean-and-mean mentality produces a manageable, reliable system that users can understand. The argument for the latter is that many users want lots of features. In any event, it should also be clear that any students planning to write a full-blown, state-of-the-art operating system from scratch have their work cut out for them.

Although Windows 2000 is already the world heavyweight champion in terms of pure mass, it is still growing, with bugs being fixed and new features being added. The way Microsoft manages its development is worth noting. Hundreds

Year	AT&T	BSD	MINIX	Linux	Solaris	Win NT
1976	V6 9K					
1979	V7 21K					
1980		4.1 38K				
1982	Sys III 58K					
1984		4.2 98K				
1986		4.3 179K				
1987	SVR3 92K		1.0 13K			
1989	SVR4 280K					
1991				0.01 10K		
1993		Free 1.0 235K			5.3 850K	3.1 6M
1994		4.4 Lite 743K		1.0 165K		3.5 10M
1996				2.0 470K		4.0 16M
1997			2.0 62K		5.6 1.4M	
1999				2.2 1M		
2000		Free 4.0 1.4M			5.8 2.0M	2000 29M

Figure 11-3. A comparison of some operating system sizes. The first string in each box is the version; the second is the size measured in lines of source code, where K = 1000 and M = 1,000,000. Comparisons within a column have real meaning; comparisons across columns do not, as discussed in the text.

of programmers work on various aspects of Windows 2000 all day. Whenever a piece of code is finished, the programmer submits it electronically to the build team. At 6 P.M. every day, the door is closed and the system is rebuilt (i.e., recompiled and linked). Each build gets a unique sequence number, which can be seen by examining the version number of *ntoskrnl.exe* (the first public release of Windows 2000 was build 2195).

The new operating system is electronically distributed to thousands of machines around the Microsoft campus in Redmond, WA, where it is subjected to intense stress tests all night. Early the next morning, the results of all the tests are sent to the relevant groups, so they can see if their new code works. Each team then decides which code they want to work on that day. During the day, the programmers work on their code and at 6 P.M. the build-and-test-cycle begins anew.

11.2 PROGRAMMING WINDOWS 2000

It is now time to start our technical study of Windows 2000. However, before getting into the details of the internal structure, we will first take a look at the programming interface and the registry, a small in-memory data base.

11.2.1 The Win32 Application Programming Interface

Like all other operating systems, Windows 2000 has a set of system calls it can perform. However, Microsoft has never made the list of Windows system calls public, and it also changes them from release to release. Instead, what Microsoft has done is define a set of function calls called the **Win32 API** (**Win32 Application Programming Interface**) that are publicly known and fully documented. These are library procedures that either make system calls to get the work done, or, in some cases, do the work right in user space. The existing Win32 API calls do not change with new releases of Windows, although new API calls are added frequently.

Binary programs for the Intel x86 that adhere exactly to the Win32 API interface will run unmodified on all versions of Windows since Windows 95. As shown in Fig. 11-4, an extra library is needed for Windows 3.x to match a subset of the 32-bit API calls to the 16-bit operating system, but for the other systems no adaptation is needed. It should be noted that Windows 2000 adds substantial new functionality to Win32, so it has additional API calls not included on older versions of Win32 and which will not work on older versions of Windows.

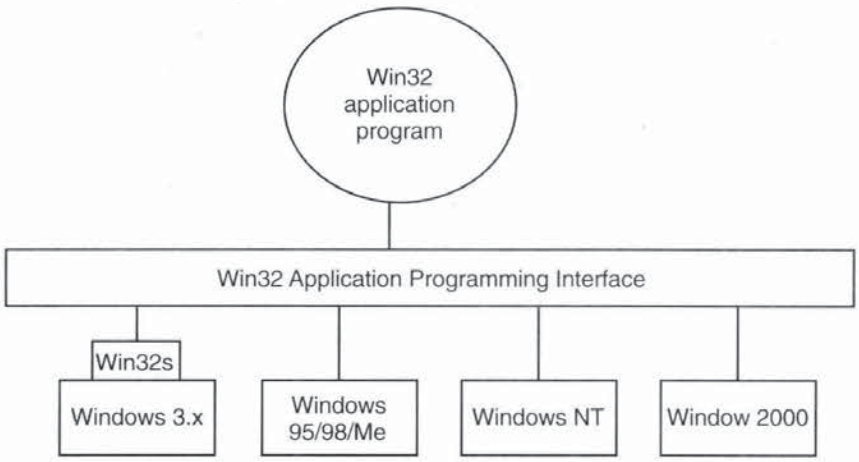


Figure 11-4. The Win32 API allows programs to run on almost all versions of Windows.

The Win32 API philosophy is completely different from the UNIX philosophy. In the latter, the system calls are all publicly known and form a minimal interface: removing even one of them would reduce the functionality of the operating system. The Win32 philosophy is to provide a very comprehensive interface, often with three or four ways of doing the same thing, and including many functions (i.e., procedures) that clearly should not be (and are not) system calls, such as an API call to copy an entire file.

Many Win32 API calls create kernel objects of one kind or another, including files, processes, threads, pipes, and so on. Every call creating an object returns a

result called a **handle** to the caller. This handle can subsequently be used to perform operations on the object. Handles are specific to the process that created the object referred to by the handle. They cannot be passed directly to another process and used there (just as UNIX file descriptors cannot be passed to other processes and used there). However, under certain circumstances, it is possible to duplicate a handle and pass it to other processes in a protected way, allowing them controlled access to objects belonging to other processes. Every object also has a security descriptor associated with it, telling in detail who may and may not perform what kinds of operations on the object.

Not all system-created data structures are objects and not all objects are kernel objects. The only ones that are true kernel objects are those that need to be named, protected, or shared in some way. Every kernel object has a system-defined type, has well-defined operations on it, and occupies storage in kernel memory. Although users can perform the operations (by making Win32 calls), they cannot get at the data directly.

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. For example, when a device driver is loaded, an object is created for it holding its properties and pointers to the functions it contains. Within the operating system, the driver is then referred to by using its object.

Windows 2000 is sometimes said to be object-oriented because the only way to manipulate objects is by invoking operations on their handles by making Win32 API calls. On the other hand, it lacks some of the most basic properties of object-oriented systems such as inheritance and polymorphism.

The Win32 API calls cover every conceivable area an operating system could deal with, and quite a few it arguably should not deal with. Naturally, there are calls for creating and managing processes and threads. There are also many calls that relate to interprocess (actually, interthread) communication, such as creating, destroying, and using mutexes, semaphores, events, and other IPC objects.

Although much of the memory management system is invisible to programmers (fundamentally, it is just demand paging), one important feature is visible: namely the ability of a process to map a file onto a region of its virtual memory. This allows the process the ability to read and write parts of the file as though they were memory words.

An important area for many programs is file I/O. In the 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, and much more.

Another area for which Win32 provides calls is security. Every process has an ID telling who it is and every object can have an 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 a fine-grained security in which specific individuals can be allowed or denied specific access to every object.

Processes, threads, synchronization, memory management, file I/O, and security system calls are nothing new. Other operating systems have them too, although generally not hundreds of them, as Win32 does. But what really distinguishes Win32 are the thousands upon thousands of calls for the graphical interface. 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 input devices as well as audio, printing, and other output devices. In short, the Win32 API (especially the GUI part) is immense and we could not even begin to describe it in any detail in this chapter, so we will not try. Interested readers should consult one of the many books on Win32 (e.g., Petzold, 1999; Simon, 1997; and Rector and Newcomer, 1997).

Although the Win32 API is available on Windows 98 (as well as on the consumer electronics operating system, Windows CE), not every version of Windows implements every call and sometimes there are minor differences as well. For example, Windows 98 does not have any security, so those API calls that relate to security just return error codes on Windows 98. Also, Windows 2000 file names use the Unicode character set, which is not available on Windows 98 and Windows 98 file names are not case sensitive, whereas Windows 2000 file names are case sensitive (although some kinds of searches on file names are not case sensitive). There are also differences in parameters to some API function calls. On Windows 2000, for example, all the screen coordinates given in the graphics functions are true 32-bit numbers; on Windows 98, only the low-order 16 bits are used because much of the graphics subsystem is still 16-bit code. The existence of the Win32 API on several different operating systems makes it easier to port programs between them, but since these minor variations exist, some care must be taken to achieve portability.

11.2.2 The Registry

Windows needs to keep track of a great deal of information about hardware, software, and users. In Window 3.x, this information was stored in hundreds of *.ini* (initialization) files spread all over the disk. Starting with Windows 95, nearly all the information needed for booting and configuring the system and tailoring it to the current user was gathered in a big central database called the **registry**. In this section we will give an overview of the Windows 2000 registry.

To start with, it is worth noting that although many parts of Windows 2000 are complicated and messy, the registry is one of the worst, and the cryptic nomenclature does not make it any better. Fortunately, entire books have been

written describing it (Born, 1998; Hipson, 2000; and Ivens, 1998). That said, the *idea* behind the registry is very simple. It consists of a collection of directories, each of which contains either subdirectories or entries. In this respect it is a kind of file system for very small files. It has directories and entries (the files).

The confusion starts with the fact that Microsoft calls a directory a **key**, which it definitely is not. Furthermore, all the top-level directories start with the string *HKEY*, which means handle to key. Subdirectories tend to have somewhat better chosen names, although not always.

At the bottom of the hierarchy are the entries, called **values**, which contain the information. Each value has three parts: a name, a type, and the data. The name is just a Unicode string, often default if the directory contains only one value. The type is one of 11 standard types. The most common ones are Unicode string, a list of Unicode strings, a 32-bit integer, an arbitrary length binary number, and a symbolic link to a directory or entry elsewhere in the registry. Symbolic names are completely analogous to symbolic links in file systems or shortcuts on the Windows desktop: they allow one entry to point to another entry or directory. A symbolic link can also be used as a key, meaning that something that appears to be a directory is just a pointer to a different directory.

At the top level, the Windows 2000 registry has six keys, called **root keys**, as listed in Fig. 11-5. Some interesting **subkeys** (subdirectories) are also shown here. To see this list on your system, use one of the registry editors, either *regedit* or *regedt32*, which unfortunately display different information and use different formats. They can also change registry values. Amateurs should not change keys or values on any system they plan to boot again. Just looking is safe, though. You have been warned.

The first key (i.e., directory), HKEY_LOCAL_MACHINE, is probably the most important as it contains all the information about the local system. It has five subkeys (i.e., subdirectories). The HARDWARE subkey contains many subkeys telling all about the hardware and which driver controls which piece of hardware. This information is constructed on the fly by the plug-and-play manager as the system boots. Unlike the other subkeys, it is not stored on disk.

The SAM (Security Account Manager) subkey contains the user names, groups, passwords, and other account and security information needed for logging in. The SECURITY subkey contains general security policy information, such as minimum length for passwords, how many failed login attempts are tolerated before the police are called, etc.

The SOFTWARE subkey is where software manufacturers store preferences etc. For example, if a user has Adobe *Acrobat*, *Photoshop* and *Premiere* installed, there will be a subkey Adobe here, and below that further subkeys for Acrobat, Photoshop, Premiere, and any other Adobe products. The entries in these subdirectories can store anything the Adobe programmers want to put there, generally system-wide properties such as the version and build number, how to uninstall the package, drivers to use, and so forth. The registry saves them the trouble of hav-

Key	Description
HKEY_LOCAL_MACHINE	Properties of the hardware and software
HARDWARE	Hardware description and mapping of hardware to drivers
SAM	Security and account information for users
SECURITY	System-wide security policies
SOFTWARE	Generic information about installed application programs
SYSTEM	Information for booting the system
HKEY_USERS	Information about the users; one subkey per user
USER-AST-ID	User AST's profile
AppEvents	Which sound to make when (incoming email/fax, error, etc.)
Console	Command prompt settings (colors, fonts, history, etc.)
Control Panel	Desktop appearance, screensaver, mouse sensitivity, etc.
Environment	Environment variables
Keyboard Layout	Which keyboard: 102-key US, AZERTY, Dvorak, etc.
Printers	Information about installed printers
Software	User preferences for Microsoft and third party software
HKEY_PERFORMANCE_DATA	Hundreds of counters monitoring system performance
HKEY_CLASSES_ROOT	Link to HKEY_LOCAL_MACHINE\SOFTWARE\CLASSES
HKEY_CURRENT_CONFIG	Link to the current hardware profile
HKEY_CURRENT_USER	Link to the current user profile

Figure 11-5. The root keys registry keys and selected subkeys. The capitalization has no meaning but follows the Microsoft practice here.

ing to invent their own method for storing this information. User-specific information also goes in the registry, but under HKEY_USERS.

The SYSTEM subkey holds mostly information about booting the system, for example, the list of drivers that must be loaded. It also holds the list of services (daemons) that must be started after booting up and the configuration information for all of them.

The next top-level key is HKEY_USERS, which contains the profiles of all the users. All the user-specific preferences in a number of areas are stored here. When a user changes a preference using the control panel, for example, the desktop color scheme, the new settings are recorded here. In fact, many of the programs on the control panel do little more than collect user information and change the registry accordingly. Some of the subkeys under HKEY_USERS are shown in Fig. 11-5 and should need little additional comment. Some of the subkeys, such as Software, contain surprisingly large numbers of subkeys, even if no software packages are installed.

The next top-level key, HKEY_PERFORMANCE_DATA contains neither data read in from the disk nor data collected by the plug-and-play manager. Instead, it offers a window into the operating system. The system itself contains hundreds of counters for monitoring system performance. These counters are accessible via

this registry key. When a subkey is queried, a specified procedure is run to collect and return the information (possibly by reading one or more counters and combining them in some way). This key is not visible using *regedit* or *regedt32*. Instead one has to use the performance tools, such as *pfmon*, *perfmon*, and *pview*. There are many such tools, some on the Windows 2000 CD-ROM, some in the resource kits, and some from third parties.

The next three top-level keys do not actually exist. Each one is a symbolic link to some place elsewhere in the registry. The HKEY_CLASSES_ROOT key is the most interesting. It points to the directory that handles COM (Component Object Model) objects and also the associations between file extensions and programs. When a user double clicks on a file ending in, say, *.doc*, the program catching the mouse click looks here to see which program to run (probably *Microsoft Word*). The complete database of recognized extensions and which program each one is owned by is under this key.

The HKEY_CURRENT_CONFIG key links to the current hardware configuration. A user can construct multiple hardware configurations, for example by disabling various devices to see if they were the cause of strange system behavior. This key points to the current configuration. Similarly, HKEY_CURRENT_USER points to the current user so that user's preferences can be found quickly.

None of the last three keys really adds anything, since the information was available elsewhere anyway (although less conveniently accessible). Thus despite the fact that *regedit* and *regedt32* list five top-level keys, there are really only three top-level directories and one of those is not shown among the five displayed.

The registry is fully available 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. 11-6.

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 11-6. Some of the Win32 API calls for using the registry

When the system is turned off, most of the registry information (but not all, as discussed above) is stored on the disk in files called **hives**. Most of them are in *\winnt\system32\config*. Because their integrity is so critical to correct system functioning, when they are updated, backups are made automatically and writes are done using atomic transactions to prevent corruption in the event of system crash during the write. Loss of the registry requires reinstalling all software.

11.3 SYSTEM STRUCTURE

In the previous sections we examined Windows 2000 as seen by the programmer. 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. Although there are many books on how to use Windows 2000, there are many fewer on how it works. Far and away the best place to look for additional information on this topic is *Inside Windows 2000*, 3rd ed. by Solomon and Russinovich (2000). Some of the material in this chapter is based on information in that book and information from the authors. Microsoft was also a key source.

11.3.1 Operating System Structure

Windows 2000 consists of two major pieces: the operating system itself, which runs in kernel mode, and the environment subsystems, which run in user mode. The kernel is a traditional kernel in the sense that it handles process management, memory management, file systems, and so on. The environment subsystems are somewhat unusual in that they are separate processes that help user programs carry out certain system functions. In the following sections we will examine each of these parts in turn.

One of NT's many improvements over Windows 3.x was its modular structure. It consisted of a moderately small kernel that ran in kernel mode, plus some server processes that ran in user mode. User processes interacted with the server processes using the client-server model: a client sent a request message to a server, and the server did the work and returned the result to the client via a second message. This modular structure made it easier to port it to several computers besides the Intel line, including the DEC Alpha, IBM PowerPC, and SGI MIPS. It also protected the kernel from bugs in the server code. However, for performance reasons, starting with NT 4.0, pretty much all of the operating system (e.g., system call handling and all of the screen graphics) was put back into kernel mode. This design was carried over to Windows 2000.

Nevertheless, there is still some structure in Windows 2000. It is divided into several layers, each one using the services of the ones beneath it. The structure is illustrated in Fig. 11-7. One of the layers is divided horizontally into many modules. Each module has some particular function and a well-defined interface to the other modules.

The lowest two software layers, the HAL and the kernel, are written in C and in assembly language and are partly machine dependent. The upper ones are written entirely in C and are almost entirely machine independent. The drivers are written in C, or in a few cases C++. Below we will first examine the various components of the system starting at the bottom and working our way up.

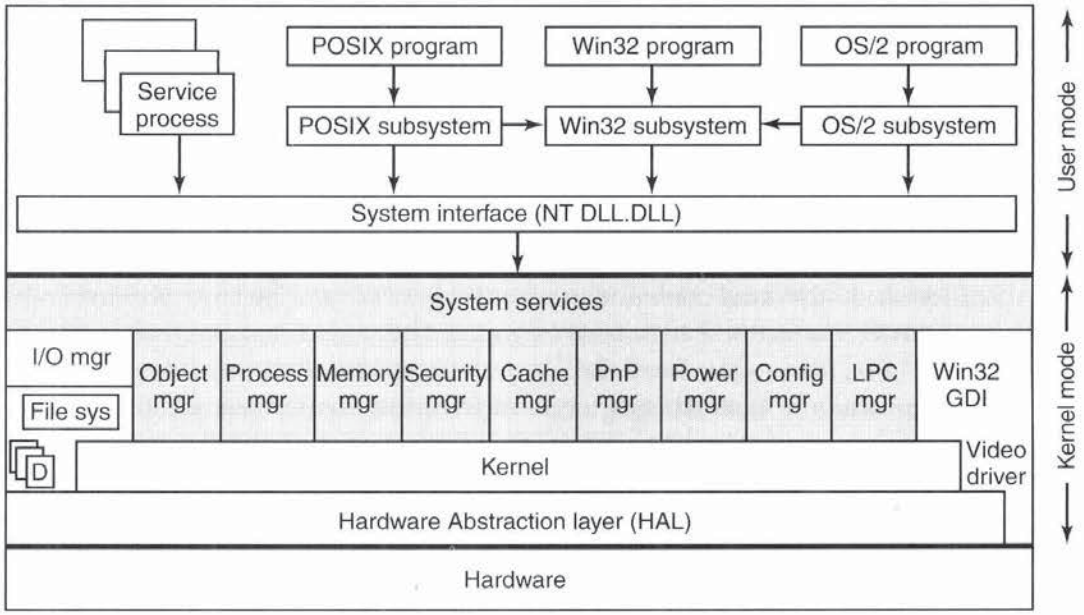


Figure 11-7. The structure of Windows 2000 (slightly simplified). The shaded area is the executive. The boxes indicated by D are device drivers. The service processes are system daemons.

The Hardware Abstraction Layer

One of the goals of Windows 2000 (and Windows NT before it) was to make the operating system portable across platforms. Ideally, when a new machine comes along, it should be possible to just recompile the operating system with the new machine's compiler and have it run the first time. Unfortunately, life is not like that. While the upper layers of the operating system can be made completely portable (because they mostly deal with internal data structures), the lower layers deal with device registers, interrupts, DMA, and other hardware features that differ appreciably from machine to machine. Even though most of the low-level code is written in C, it cannot just be scooped up from a Pentium, plopped down on, say, an Alpha, recompiled, and rebooted due to the many small hardware differences between the Pentium and the Alpha that have nothing to do with the different instruction sets and which cannot be hidden by the compiler.

Fully aware of this problem, Microsoft made a serious attempt to hide many of the machine dependencies in a thin layer at the bottom called the **HAL** (**Hardware Abstraction Layer**). (The name HAL was no doubt inspired by the computer HAL in the late Stanley Kubrick's movie *2001: A Space Odyssey*. Rumor has it that Kubrick chose the name "HAL" by taking the name of the then-dominant computer company—IBM— and subtracting 1 from each letter.)

The job of the HAL is to present the rest of the operating system with abstract hardware devices, in particular, devoid of the warts and idiosyncracies with which

real hardware is so richly endowed. These devices are presented in the form of machine-independent services (procedure calls and macros) that the rest of the operating system and the drivers can use. By using the HAL services (which are identical on all Windows 2000 systems, no matter what the hardware is) and not addressing the hardware directly, drivers and the kernel require fewer changes when being ported to new hardware. 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, namely, implement all of the HAL services.

The services chosen for inclusion in the HAL are those that relate to the chip set on the parentboard and which vary from machine to machine within reasonably predictable limits. In other words, it is designed to hide the differences between one vendor's parentboard and another one's, but not the differences between a Pentium and an Alpha. The HAL services include access to the device registers, bus-independent device addressing, interrupt handling and resetting, DMA transfers, control of the timers and real-time clock, low-level spin locks and multiprocessor synchronization, interfacing with the BIOS and its CMOS configuration memory. The HAL does not provide abstractions or services for specific I/O devices such as keyboards, mice, or disks or for the memory management unit.

As an example of what the hardware abstraction layer does, consider the issue of memory-mapped I/O versus 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 often 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 (ISA, PCI, SCSI, USB, 1394, etc.), it can happen that two or more devices have the same bus address, so some way is needed to distinguish them. The HAL provides a service for identifying devices by mapping bus-relative device addresses onto system-wide logical addresses. In this way, drivers are not required to keep track of which device is on which bus.

These logical addresses are analogous to the handles the operating system gives user programs to refer to files and other system resources. 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 system-wide way and also provides services 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 system-wide DMA engine and DMA engines on specific I/O cards can be handled. Devices are referred to by their logical addresses. The HAL also 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 nsec starting at 1 January 1601, which is far more precise than MS-DOS's keeping track of time in units of 2 sec since 1 January 1980 and provides support for the many computer-related activities in the 17th, 18th, and 19th centuries. 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 some 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 only held for a few machine instructions.

Finally, after the system has been booted, the HAL talks to the BIOS and inspects the CMOS configuration memory, if any, 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 so other system components can look it up without having to understand how the BIOS or configuration memory work. A summary of some of the things the HAL does is given in Fig. 11-8.

Since the HAL is highly-machine dependent, it must match the system it is installed on perfectly, so a variety of HALs are provided on the Windows 2000 CD-ROM. At system installation time, the appropriate one is selected and copied to the system directory *\winnt\system32* on the hard disk as *hal.dll*. All subsequent boots use this version of the HAL. Removing this file will make the system unbootable.

Although the HAL is reasonably efficient, for multimedia applications, it may not be fast enough. For this reason, Microsoft also produced a software package called **DirectX**, which augments the HAL with additional procedures and allows user processes much more direct access to the hardware. DirectX is somewhat specialized, so we will not discuss it further in this chapter.

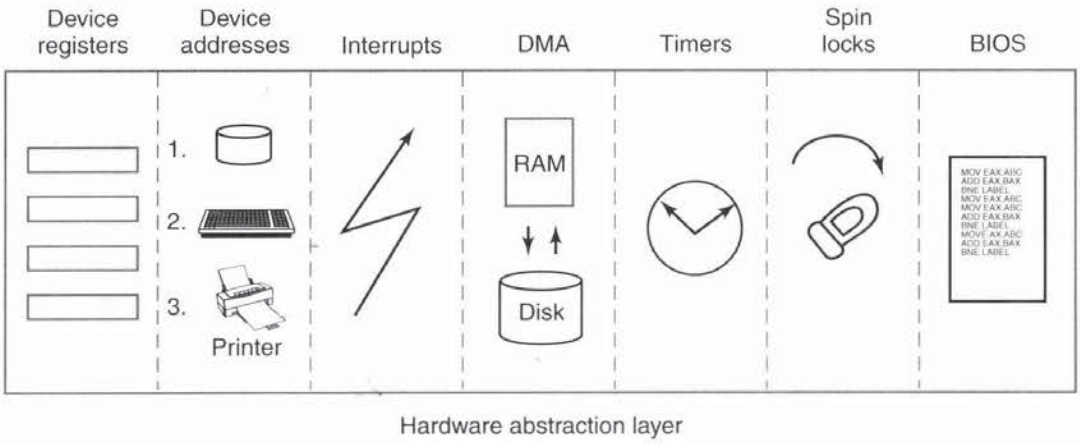


Figure 11-8. Some of the hardware functions the HAL manages.

The Kernel Layer

Above the hardware abstraction layer is a layer that contains what Microsoft calls the **kernel**, as well as the device drivers. Some early documentation refers to the kernel as the "microkernel," which it really never was because the memory manager, file system, and other major components resided in kernel space and ran in kernel mode from day one. The kernel is certainly not a microkernel now since virtually the entire operating system was put in kernel space starting with NT 4.0.

In the chapter on UNIX, we used the term "kernel" to mean everything running in kernel mode. In this chapter we will (very grudgingly) reserve the term "kernel" for the part labeled as such in Fig. 11-7 and call the totality of the code running in kernel mode the "operating system." Part of kernel (and much of the HAL) is permanently resident in main memory (i.e., is not paged). By adjusting its priority, it can control whether it can tolerate being preempted by I/O interrupts or not. Although a substantial fraction of the kernel is machine specific, most of it is nevertheless written in C, except where top performance overshadows all other concerns.

The purpose of the kernel is to make the rest of the operating system completely independent of the hardware, and thus highly portable. It picks up where the HAL leaves off. It accesses the hardware via the HAL and builds upon the extremely low-level HAL services to construct higher-level abstractions. For example, the HAL has calls to associate interrupt service procedures with interrupts, and set their priorities, but does little else in this area. The kernel, in contrast, provides a complete mechanism for doing context switches. It properly saves all the CPU registers, changes the page tables, flushes the CPU cache, and so on, so that when it is done, the previously running thread has been saved in tables in memory. It then sets up the new thread's memory map and loads its registers so the new thread can start running.

The code for thread scheduling is also in the kernel. When it is time to see if a new thread can run, for example, after a quantum runs out or after an I/O interrupt completes, the kernel chooses the thread and does the context switch necessary to run it. From the point of view of the rest of the operating system, thread switching is automatically handled by lower layers without any work on their part and in a portable way. 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 also has another key function: providing low-level support for two classes of objects: control objects and dispatcher objects. These objects are not the objects that user processes get handles to, but are internal objects upon which the executive builds the user objects.

Control objects are those objects that control the system, including primitive process objects, interrupt objects, and two somewhat strange objects called DPC and APC. A **DPC** (**Deferred Procedure Call**) object is used to split off the non-time-critical part of an interrupt service procedure from the time critical part. Generally, an interrupt service procedure saves a few volatile hardware registers associated with the interrupting I/O device so they do not get overwritten and reenables the hardware, but saves the bulk of the processing for later.

For example, after a key is struck, the keyboard interrupt service procedure reads the key code from a register and reenables the keyboard interrupt, but does not need to process the key immediately, especially if something more important (i.e., higher priority) is currently going on. As long as the key is processed within about 100 msec, the user will be none the wiser. DPCs are also used for timer expirations and other activities whose actual processing need not be instantaneous. The DPC queue is the mechanism for remembering that there is more work to do later.

Another kernel control object is the **APC** (**Asynchronous Procedure Call**). APCs are like DPCs except that they execute in the context of a specific process. When processing a key press, it does not matter whose context the DPC runs in because all that is going to happen is that the key code will be inspected and probably put in a kernel buffer. However, if an interrupt requires copying a buffer from kernel space to a buffer in some user process' address space (e.g., as it may on completion of a read from the modem), then the copying procedure needs to run in the receiver's context. The receiver's context is needed so the page table will contain both the kernel buffer and the user buffer (all processes contain the entire kernel in their address spaces, as we will see later). For this reason, the kernel distinguishes between DPCs and APCs.

The other kind of kernel objects are **dispatcher objects**. These include semaphores, mutexes, events, waitable timers, and other objects that threads can wait on. The reason that these have to be handled (in part) in the kernel is that they are intimately intertwined with thread scheduling, which is a kernel task. As a little aside, mutexes 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 the Windows 2000 designers considered bizarre. (The OS/2 semantics are relevant because NT was originally conceived of as a replacement for OS/2, the operating system shipped on IBM's PC/2.)

The Executive

Above the kernel and device drivers is the upper portion of the operating system, called the **executive**, shown as the shaded area in Fig. 11-7. The executive is written in C, is architecture independent, and can be ported to new machines with relatively little effort. It consists of 10 components, each of which is just a collection of procedures that work together to accomplish some goal. There are no hard boundaries between the pieces and different authors describing the executive might even group the procedures differently into components. It should be noted that components on the same level can (and do) call each other extensively.

The **object manager** manages all objects known to the operating system. These include processes, threads, files, directories, semaphores, I/O devices, timers, and many others. The object manager allocates a block of virtual memory from kernel address space when an object is created and returns it to the free list when the object is deallocated. Its job is to keep track of all the objects.

To avoid any confusion, most of the executive components labeled "manager" in Fig. 11-7 are not processes or threads, but merely collections of procedures that other threads can execute when in kernel mode. A few of them, such as the power manager and plug-and-play manager, really are independent threads though.

The object manager also manages a name space in which newly created objects may be placed so they can be referred to later. All other components of the executive use objects heavily to do their work. Objects are so central to the functioning of Windows 2000 that they will be discussed in detail in the next section.

The **I/O manager** provides a framework for managing I/O devices and provides generic I/O services. It provides the rest of the system with device-independent I/O, calling the appropriate driver to perform physical I/O. It is also home to all the device drivers (indicated by *D* in Fig. 11-7). The file systems are technically device drivers under control of the I/O manager. Two different ones are present for the FAT and NTFS file systems, each one independent of the others and controlling different disk partitions. All the FAT file systems are managed by a single driver. We will study I/O further in Sec. 11.6 and one of the file systems, NTFS, in Sec. 11.7.

The **process manager** handles processes and threads, including their creation and termination. It deals with the mechanisms used to manage them, rather than policies about how they are used. It builds upon the kernel process and thread objects and adds extra functionality to them. It is the key to multiprogramming in Windows 2000. We will study process and thread management in Sec. 11.4.

The **memory manager** implements Windows 2000's demand-paged virtual memory architecture. It manages the mapping of virtual pages onto physical page frames. It thereby enforces the protection rules that restrict each process to only access those pages belonging to its address space and not to other processes' address spaces (except under special circumstances). It also handles certain system calls that relate to virtual memory. We will study memory management in Sec. 11.5.

The **security manager** enforces Windows 2000's elaborate security mechanism, which meets the U.S. Dept. of Defense's Orange Book C2 requirements. The Orange Book specifies a large number of rules that a conforming system must meet, starting with authenticated login through how access control is handled, to the fact that virtual pages must be zeroed out before being reused. We will study the security manager in Sec. 11.8.

The **cache manager** keeps the most recently used disk blocks in memory to speed up access to them in the (likely) event that they are needed again. Its job is to figure out which blocks are probably going to be needed again and which ones are not. It is possible to configure Windows 2000 with multiple file systems, in which case the cache manager works for all of them, so each one does not have to do its own cache management. When a block is needed, the cache manager is asked to supply it. If it does not have the block, the cache manager calls upon the appropriate file system to get it. Since files can be mapped into processes' address spaces, the cache manager must interact with the virtual memory manager to provide the necessary consistency. The amount of space devoted to caching is dynamic and can increase or decrease as demands on it change. We will study the cache manager in Sec. 11.9.

The **plug-and-play manager** is sent all notifications of newly attached devices. For some devices, a check is made at boot time and not thereafter. Other devices, for example, USB devices, can be attached at any time and their attachment triggers a message to the plug-and-play manager, which then locates and loads the appropriate driver.

The **power manager** rides herd on power usage. This consists of turning off the monitor and disks after they have been idle for a while. On laptops, the power manager monitors battery usage and takes action when the battery is about to run dry. Such action typically tells programs to save their files and prepare for a graceful shutdown. And pronto.

The **configuration manager** is in charge of the registry. It adds new entries and looks up keys when asked to.

The **local procedure call manager** provides for a highly-efficient interprocess communication used between processes and their subsystems. Since this path is needed to carry out some system calls, efficiency is critical here, which is why the standard interprocess communication mechanisms are not used.

The Win32 GDI executive module handles certain system calls (but not all of them). It was originally in user space but was moved to kernel space in NT 4.0 to

improve performance. The **GDI** (**Graphics Device Interface**) handles image management for the monitor and printers. It provides system calls to allow user programs to write on the monitor and printers in a device-independent way. It also contains the window manager and display driver. Prior to NT 4.0, it, too, was in user space but the performance was disappointing, so Microsoft moved it into the kernel to speed it up. It is worth mentioning that Fig. 11-7 is not at all to scale. For example, the Win32 and graphics device interface module is larger than the rest of the executive combined.

At the top of the executive is a thin layer called **system services**. Its function is to provide an interface to the executive. It accepts the true Windows 2000 system calls and calls other parts of the executive to have them executed.

At boot time, Windows 2000 is loaded into memory as a collection of files. The main part of the operating system, consisting of the kernel and executive, is located in the file *ntoskrnl.exe*. The HAL is a shared library located in a separate file, *hal.dll*. The Win32 and graphics device interface are together in a third file, *win32k.sys*. Finally, many device drivers are also loaded. Most of these have extension *.sys*.

Actually, things are not quite that simple. The *ntoskrnl.exe* file comes in uniprocessor and multiprocessor versions. Also, there are versions for the Xeon processor, which can have more than 4 GB of physical memory and the Pentium, which cannot. Finally, versions can consist of a free build (sold in stores and preinstalled by computer manufacturers) or a checked build (for debugging purposes). Together there could be eight combinations, although two pairs were combined leaving only six. One of these is copied to *ntoskrnl.exe* when the system is installed.

The checked builds are worth a few words. When a new I/O device is installed on a PC, there is invariably a manufacturer-supplied driver that has to be installed to make it work. Suppose that an IEEE 1394 card is installed on a computer and appears to work fine. Two weeks later the system suddenly crashes. Who does the owner blame? Microsoft.

The bug may indeed be Microsoft's, but some bugs are actually due to flakey drivers, over which Microsoft has no control and which are installed in kernel memory and have full access to all kernel tables as well as the entire hardware. In an attempt to reduce the number of irate customers on the phone, Microsoft tries to help driver writers debug their code by putting statements of the form

```
ASSERT(some condition)
```

throughout the code. These statements make sanity checks on all parameters to internal kernel procedures (which may be freely called by drivers) and make many other checks as well. The free builds have *ASSERT* defined as a macro that does nothing, removing all the checks. The checked builds have it defined as

```
#define ASSERT(a) if (! (a)) error( ... )
```

causing all the checks to appear in the *ntoskrnl.exe* executable code and be carried out at run time. While this slows down the system enormously, it helps driver writers debug their drivers before they ship them to customers. The checked builds also have numerous other debugging features turned on.

The Device Drivers

The last part of Fig. 11-7 consists of the **device drivers**. Each device driver can control one or more I/O devices, but a device driver can also do things not related to a specific device, such as encrypting a data stream or even just providing access to kernel data structures. Device drivers are not part of the *ntoskrnl.exe* binary. The advantage of this approach is that once a driver has been installed on a system, it is added to a list in the registry and is loaded dynamically when the system boots. In this way, *ntoskrnl.exe* is the same for everyone, but every system is configured precisely for those devices it contains.

There are device drivers for macroscopically visible I/O devices such as disks and printers, but also for many internal devices and chips that practically no one has ever heard of. In addition, the file systems are also present as device drivers, as mentioned above. The largest device driver, the one for Win32, GDI, and video, is shown on the far right of Fig. 11-7. It handles many system calls and most of the graphics. Since customers can install new device drivers, they have the power to affect the kernel and corrupt the system. For this reason, drivers must be written with great care.

11.3.2 Implementation of Objects

Objects are probably the single most important concept in Windows 2000. They provide a uniform and consistent interface to all system resources and data structures such as processes, threads, semaphores, etc. This uniformity has various facets. First, all objects are named and accessed the same way, using object handles. Second, because all accesses to objects go through the object manager, it is possible to put all the security checks in one place and ensure that no process can make an end run around them. Third, sharing of objects among processes can be handled in a uniform way. Fourth, since all object opens and closes go through the object manager, it is easy to keep track of which objects are still in use and which can be safely deleted. Fifth, this uniform model for object management makes it easy to manage resource quotas in a straightforward way.

A key to understanding objects is to realize that an (executive) object is just some number of consecutive words in memory (i.e., in kernel virtual address space). An object is a data structure in RAM, no more and no less. A file on disk is not an object, although an object (i.e., a data structure in kernel virtual address space) is created for a file when it is opened. 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. In fact, when the system boots, there are no objects present at all (except for the idle and system processes, whose objects are hardwired into the *ntoskrnl.exe* file). All other objects are created on the fly as the system boots up and various initialization (and later user) programs run.

Objects have a structure, as shown in Fig. 11-9. 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 object space, security information (so a check can be made when an object is opened), and a list of processes with open handles to the object (if a certain debugging flag is enabled).

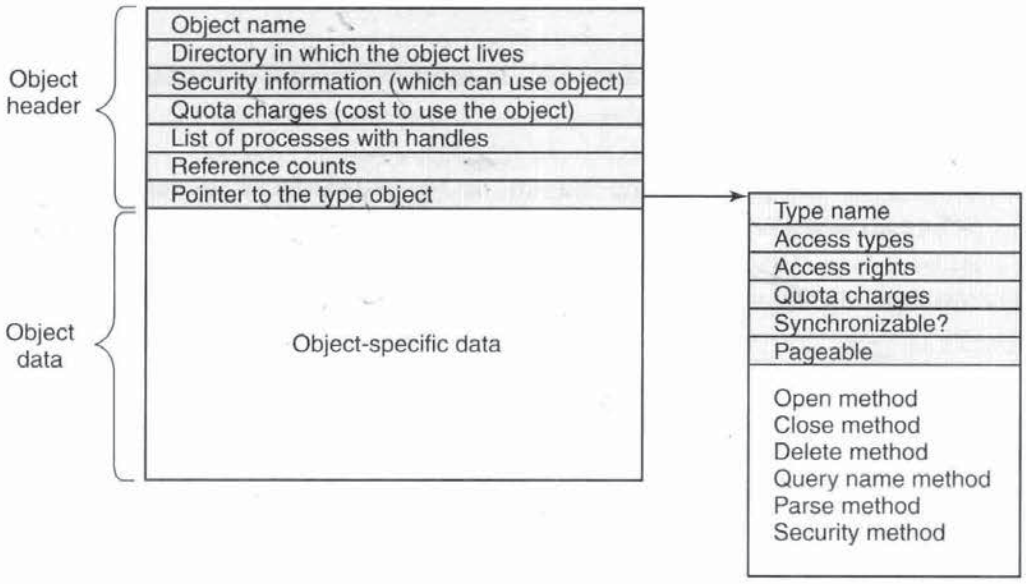


Figure 11-9. The structure of an object.

Each object header also contains a quota charge field, which is the charge levied against a process for opening the object. If a file object costs 1 point and a process belongs to a job that has 10 file points worth of quota, the processes in that job can only open 10 files in total. In this way resource limits can be enforced for each object type separately.

Objects occupy valuable real estate—pieces of kernel virtual address space—so when an object is no longer needed it should be removed and its address space reclaimed. The mechanism for reclamation is to have a reference counter in each object's header. It counts the number of open handles held by processes. This counter is incremented every time the object is opened and decremented every time it is closed. When it hits 0, no more users hold handles to the object. When an object is acquired or released by an executive component, a second counter is incremented or decremented, even though no actual handle is issued. When both counters hit 0, no user process is using the object and no executive process is using the object, so the object can be removed and its memory freed.

The object manager needs to maintain dynamic data structures (its objects), but it is not the only part of the executive with this need. Other pieces also need to allocate and release chunks of kernel memory dynamically. To meet these needs, the executive maintains two page pools in kernel address space: for objects and for other dynamic data structures. Such pools operate as heaps, similar to the C language calls *malloc* and *free* for managing dynamic data. One pool is paged and the other is nonpaged (pinned in memory). Objects that are needed often are kept in the nonpaged pool; objects that are rarely accessed, such as registry keys and some security information, are kept in the paged pool. When memory is tight, the latter can be paged out and faulted back on demand. In fact, substantial portions of the operating system code and data structures are also pageable, to reduce memory consumption. Objects that may be needed when the system is running critical code (and when paging is not permitted) must go in the nonpaged pool. When a small amount of storage is needed, a page can be taken from either pool and then broken up into units as small as 8 bytes.

Objects are typed, which means each one has certain properties common to all objects of its type. The type is indicated by a pointer in the header to a type object, as shown in Fig. 11-9. The type object information includes items such as the type name, whether a thread can wait on the object (yes for mutexes, no for open files), and whether new objects of this type go on the paged or nonpaged pool. Each object points to its type object.

The last thing a type object has is also the most important: pointers to the code for certain standard operations such as open, close, and delete. Whenever one of these operations is invoked on an object, the pointer to the type object is followed and the relevant code located and executed. This mechanism gives the system the opportunity to initialize new objects, and recover storage when they are deleted.

Executive components can create new types dynamically. There is no definitive list of object types, but some of the more common ones are listed in Fig. 11-10. Let us briefly go over the object types in Fig. 11-10. 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 all blocked threads. An event can also be set up, so after a signal has been successfully waited for, it automatically reverts to nonsignaled state, rather than staying in signaled state.

Port, timer, and queue objects also relate to communication and synchronization. Ports are channels between processes for exchanging messages. Timers provide a way to block for a specific time interval. Queues are used to notify threads that a previously started asynchronous I/O operation has completed.

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)
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	Structure used for mapping files onto virtual address space
Key	Registry key
Object directory	Directory for grouping objects within the object manager
Symbolic link	Pointer to another object by name
Device	I/O device object
Device driver	Each loaded device driver has its own object

Figure 11-10. Some common executive object types managed by object manager.

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 objects used by the memory system for handling memory-mapped files. They record which file (or part thereof) is mapped onto which memory addresses. Keys are registry keys and are used to relate names to values. Object directories are entirely local to the object manager. They provide a way to collect related objects together in exactly the same way directories work in the file system. Symbolic links are also similar to their file system counterparts: they allow a name in one part of the object name space to refer to an object in a different part of the object name space. Each known device has a device object that contains information about it and is used to refer to the device within the system. Finally, each device driver that has been loaded has an object in the object space.

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 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 child processes). These 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 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 reading it.

The handle tables for two processes and their relationships to some objects are illustrated in Fig. 11-11. In this example, process *A* has access to threads 1 and 2 and access to mutexes 1 and 2. Process *B* has access to thread 3 and mutexes 2 and 3. The corresponding entries in the handle tables hold the rights to each of these objects. For example, process *A* might have the rights to lock and unlock its mutexes, but not the right to destroy them. Note that mutex 2 is shared by both processes allowing threads in them to synchronize. The other mutexes are not shared, which might mean that the threads within process *A* use mutex 1 for their internal synchronization and the threads within process *B* use mutex 3 for their internal synchronization.

The Object Name Space

As objects are created and deleted during execution, the object manager needs a way to keep track of them. To do this job, it maintains a name space, in which all objects in the system are located. The name space can be used by a process to locate and open a handle for some other process' object, provided it has been granted permission to do so. The object name space is one of three name spaces maintained by Windows 2000. The other ones are the file system name space and the registry name space. All three are hierarchical name spaces with multiple levels of directories for organizing entries. The directory objects listed in Fig. 11-10 provide the means to implement this hierarchical name space for objects.

Since executive objects are volatile (i.e., vanish when the computer is shut down, unlike file system and registry entries), when the system boots up, there are no objects in memory and the object name space is empty. During booting, various parts of the executive create directories and then fill them with objects. For example, as the plug-and-play manager discovers devices out there, it creates a device object for each one and enters this object into the name space. When the system is fully booted, all I/O devices, disk partitions, and other interesting discoveries are in the object name space.

Not all objects get entered by the Columbus method—just go look and see what you find. Some executive components look in the registry to see what to do. A key example here is device drivers. During bootup, the system looks in the

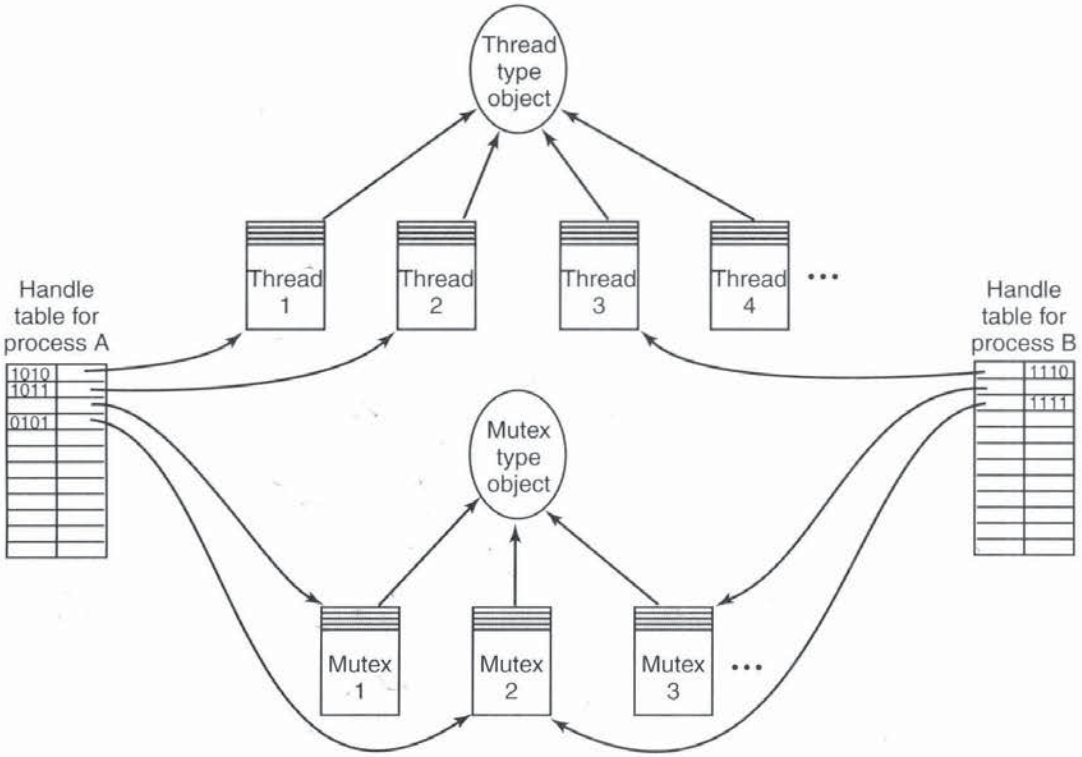


Figure 11-11. The relationship between handle tables, objects, and type objects.

registry to see which device drivers are needed. As they are loaded one by one, an object is created for each one and its name is inserted into the object space. Within the system, the driver is referred to by a pointer to its object.

Although the object name space 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 *www.sysinternals.com*. When run, this tool depicts an object name space that typically contains the object directories listed in Fig. 11-12 as well as a few others.

The somewhat 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 to speed up lookup of all path names beginning with a drive letter. The contents of the other object directories should be self explanatory.

11.3.3 Environment Subsystems

Going back to Fig. 11-7, we see that Windows 2000 consists of components in kernel mode and components in user mode. We have now completed our examination of the kernel mode components; now it is time to look at the user mode

Directory	Contents
??	Starting place for looking up MS-DOS devices like C:
Device	All discovered I/O devices
Driver	Objects corresponding to each loaded device driver
ObjectTypes	The type objects shown in Fig. 11-11
Windows	Objects for sending messages to all the windows
BaseNamedObjs	User-created 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 11-12. Some typical directories in the object name space.

components of which there are three kinds: DLLs, environment subsystems, and service processes. These components work together to provide each user process with an interface that is distinct from the Windows 2000 system call interface.

Windows 2000 supports three different documented APIs: Win32, POSIX, and OS/2. Each of these interfaces has a published list of library calls that programmers can use. The job of the DLLs (Dynamic Link Libraries) and environment subsystems is to implement the functionality of the published interface, thereby hiding the true system call interface from application programs. In particular, the Win32 interface is the official interface for Windows 2000, Windows NT, Windows 95/98/Me, and to a limited extent, Windows CE. By using the DLLs and Win32 environment subsystem, a program can be written to the Win32 specification and run unmodified on all these versions of Windows, even though the system calls are not the same on the various systems.

To see how these interfaces are implemented, let us look at Win32. A Win32 program normally contains many calls to Win32 API functions, for example, CreateWindow, DrawMenuBar, and OpenSemaphore. There are thousands of such calls, and most programs use a substantial number of them. One possible implementation would be to statically link every Win32 program with all the library procedures that it uses. If this were done, each binary program would contain one copy of each procedure it used in its executable binary.

The trouble with this approach is that it wastes memory if the user has multiple programs open at once and they use many of the same library procedures. For example, *Word*, *Excel*, and *Powerpoint* all use exactly the same procedures for opening dialog boxes, drawing windows, displaying menus, managing the clipboard, etc., so if a user had all of them open and active at once, there would be three (identical) copies of each of the libraries in memory.

To avoid this problem, all versions of Windows support shared libraries, called **DLL**s (**Dynamic Link Libraries**). Each DLL collects together a set of closely related library procedures and their data structures into a single file, usually (but not always) with extension *.dll*. When an application is linked, the linker sees that some of the library procedures belong to DLLs and records this information in the executable's header. Calls to procedures in DLLs are made indirectly through a transfer vector in the caller's address space. Initially this vector is filled with 0s, since the addresses of the procedures to be called are not yet known.

When the application process is started, the DLLs that are needed are located (on disk or in memory) and mapped into the process' virtual address space. The transfer vector is then filled in with the correct addresses so that the procedures can then be called via the transfer vector with only a negligible loss of efficiency. The win here is that even though multiple application programs have the same DLL mapped in, only one copy of the DLL text is needed in physical memory (but each process gets its own copy of the private static data in the DLL). Windows 2000 uses DLLs extremely heavily for all aspects of the system.

Now we have enough background to see how the Win32 and other process interfaces are implemented. Each user process generally links with a number of DLLs that together implement the Win32 interface. To make an API call, one of the procedures in a DLL is called, shown as step 1 in Fig. 11-13. What happens next depends on the Win32 API call. Different ones are implemented in different ways.

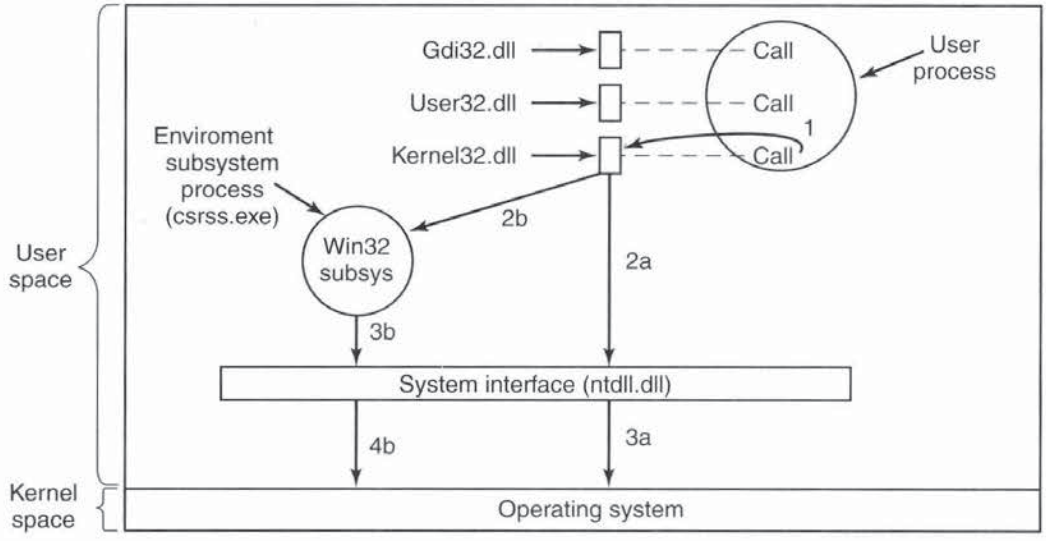


Figure 11-13. Various routes taken to implement Win32 API function calls.

In some cases, the DLL calls another DLL (*ntdll.dll*) that actually traps to the operating system. This path is shown as steps 2a and 3a in Fig. 11-13. It is also possible that the DLL does all the work itself without making a system call at all.

For other Win32 API calls a different route is taken, namely, first a message is sent to the Win32 subsystem process (*csrss.exe*), which then does some work and then makes a system call (steps 2b, 3b, and 4b). Here, too, in some cases the environment subsystem does all the work in user space and just returns immediately. The message passing between the application process and the Win32 subsystem process has been carefully optimized for performance using a special local procedure call mechanism implemented by the executive and shown as LPC in Fig. 11-7.

In the first version of Windows NT, virtually all the Win32 API calls took route 2b, 3b, 4b, putting a large chunk of the operating system in user space (e.g., the graphics). However, starting with NT 4.0, much of the code was put into kernel mode (in the Win32/GDI driver in Fig. 11-7) for performance reasons. In Windows 2000, only a small number of Win32 API calls, for example process and thread creation, take the long route. The other ones take the direct route, bypassing the Win32 environment subsystem.

As an aside, the three most important DLLs are shown in Fig. 11-13, but they are not the only ones. There are over 800 separate DLLs in the *\winnt\system32* directory totalling 130 MB. To avoid any confusion, the number of DLL *files* is over 800; the number of API calls contained in them exceeds 13,000. (The 29 million lines of code had to compile into something, after all.) A few of the more important DLLs are listed in Fig. 11-14. The number of exported functions (i.e., those visible outside the file) in each one is given, but these tend to change (meaning increase) over time. The number of exported functions in the first public release of *ntdll.dll* in Windows 2000 is 1179. These are the real system calls. The 1209 calls exported by *ntoskrnl.exe* are the functions available to device drivers and other code linked with the kernel. The list of exported functions in any *.exe* or *.dll* file can be viewed using the *depends* program in the platform SDK Kit.

File	Mode	Fcns	Contents
hal.dll	Kernel	95	Low-level hardware management, e.g., port I/O
ntoskrnl.exe	Kernel	1209	Windows 2000 operating system (kernel + executive)
win32k.sys	Kernel	-	Many system calls including most of the graphics
ntdll.dll	User	1179	Dispatcher from user mode to kernel mode
csrss.exe	User	0	Win32 environment subsystem process
kernel32.dll	User	823	Most of the core (nongraphics) system calls
gdi32.dll	User	543	Font, text, color, brush, pen, bitmap, palette, drawing, etc. calls
user32.dll	User	695	Window, icon, menu, cursor, dialog, clipboard, etc. calls
advapi32.dll	User	557	Security, cryptography, registry, management calls

Figure 11-14. Some key Windows 2000 files, the mode they run in, the number of exported function calls, and the main contents of each file. The calls in *win32k.sys* are not formally exported since *win32k.sys* is not called directly.

Although the Win32 process interface is the most important one, there are also two other ones: POSIX and OS/2. The POSIX environment provides minimal support for UNIX applications. It supports only the P1003.1 functionality and little else. It does not have threads, windowing, or networking, for example. In practice, porting any real UNIX program to Windows 2000 using this subsystem is close to impossible. It was included only because parts of the U.S. government require operating systems for government computers to be P1003.1 compliant. This subsystem is not self-contained and uses the Win32 subsystem for much of its work, but without exporting the full Win32 interface to its user programs (which would have made it usable, at no extra cost to Microsoft).

To allow UNIX users to migrate to Windows 2000, Microsoft has a product called Interix that provides a better degree of UNIX compatibility than the POSIX subsystem.

The OS/2 subsystem is similarly limited in functionality and does not support any graphical applications. In practice, it, too, is completely useless. Thus the original idea of having multiple operating system interfaces implemented by different processes in user space is essentially gone. What is left is a full Win32 implementation in kernel mode and little else.

11.4 PROCESSES AND THREADS IN WINDOWS 2000

Windows 2000 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 these concepts are implemented.

11.4.1 Fundamental Concepts

Windows 2000 supports traditional processes, which can communicate and synchronize with one another, just as they can in UNIX. Each process contains at least one thread, which in turn contains at least one fiber (lightweight thread). Furthermore, processes can be collected into jobs for certain resource management purposes. Together, jobs, processes, threads, and fibers provide a very general set of tools for managing parallelism and resources, both on uniprocessors (single-CPU machines) and on multiprocessors (multiCPU machines). A brief summary of these four concepts is given in Fig. 11-15.

Let us examine these concepts from the largest to the smallest. A **job** in Windows 2000 is a collection of one or more processes that are to be managed as a unit. In particular, there are quotas and resource limits associated with each job, stored in the corresponding job object. The quotas include items such as the maximum number of processes (prevents any process from generating an unbounded number of children), the total CPU time available to each process individually and

Name	Description
Job	Collection of processes that share quotas and limits
Process	Container for holding resources
Thread	Entity scheduled by the kernel
Fiber	Lightweight thread managed entirely in user space

Figure 11-15. Basic concepts used for CPU and resource management.

to all the processes combined, and the maximum memory usage, again, per process and total. Jobs can also impose security restrictions on the processes in the job, such as not being able to acquire administrator (superuser) power, even with the proper password.

Processes are more interesting than jobs and also more important. As in UNIX, processes are containers for resources. Every process has a 4-GB address space, with the user occupying the bottom 2 GB (optionally 3 GB on Advanced Server and Datacenter server) and the operating system occupying the rest. Thus the operating system is present in every process' address, although protected from tampering by the memory management unit hardware. A process has a process ID, one or more threads, a list of handles (managed in kernel mode), and an access token holding its security information. Processes are created using a Win32 call that takes as its input the name of an executable file, which defines the initial contents of the address space and creates the first thread.

Every process starts out with one thread, but new ones can be created dynamically. Threads forms 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 states. Threads can be created dynamically by a Win32 call that specifies the address within the enclosing process' address space it is to start running at. Every thread has a thread ID, which is taken from the same space as the process IDs, so an 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 so they can be used as byte indices into kernel tables, the same as other objects.

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. In addition to a state, an ID, and two stacks, every thread has a context (in which to save its registers when it is not running), a private area for its own local variables, and possibly its own access token. If it has its own access token, this one overrides the process access token in order to let client threads pass their access rights to server threads who are doing work for them. When a thread is finished executing, it can exit. 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 grab the handle 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.

In addition to the normal threads that run within user processes, Windows 2000 has a number of daemon threads that run only in kernel space and are not associated with any user process (they are associated with the special system or idle processes). Some perform administrative tasks, such as writing dirty pages to the disk, while others form a pool that can be assigned to a component of the executive or a driver that needs to get some work done asynchronously in the background. We will study some of these threads later when we come to memory management.

Switching threads in Windows 2000 is relatively expensive because doing a thread switch requires entering and later leaving kernel mode. To provide very lightweight pseudoparallelism, Windows 2000 provides **fibers**, which are like threads, but are scheduled in user space by the program that created them (or its run-time system). Each thread can have multiple fibers, the same way a process can have multiple threads, except that when a fiber logically blocks, it puts itself on the queue of blocked fibers and selects another fiber to run in the context of its thread.

The operating system is not aware of this transition because the thread keeps running, even though it may be first running one fiber, then another. In fact, the operating system knows nothing at all about fibers, so there are no executive objects relating to fibers, as there are for jobs, processes, and threads. There are also no true system calls for managing fibers. However, there are Win32 API calls. These are among the Win32 API calls that do not make system calls, which we mentioned during the discussion of Fig. 11-13. The relationship between jobs, processes, and threads is illustrated in Fig. 11-16.

Although we will not discuss it in much detail, Windows 2000 is capable of running on a symmetric multiprocessor system. This requirement means that the operating system code must be fully reentrant, that is, every procedure must be written in such a way that two or more CPUs may be changing its variables at once, without causing problems. In many cases this means that code sections have to be protected by spin locks or mutexes to keep additional CPUs at bay until the first one is done (i.e., serialize access to critical regions). The number of CPUs the system can handle is governed by the licensing restrictions and listed in Fig. 11-2. There is no technical reason why Windows Professional cannot run on a 32-node multiprocessor—it is the same binary as Datacenter Server, after all.

The upper limit of 32 CPUs is a hard limit because word-length bitmaps are used to keep track of CPU usage in various ways. For example, one word-length

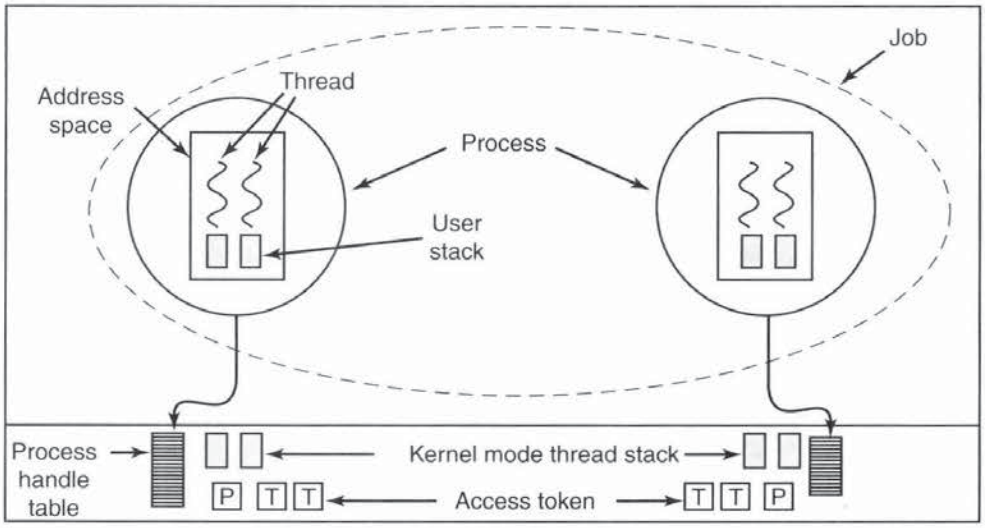


Figure 11-16. The relationship between jobs, processes, and threads. Several fibers can also be multiplexed on one thread (not shown).

bitmap keeps track of which of the (up to) 32 CPUs are currently idle, and another bitmap is used per process to list the CPUs this process is permitted to run on. The 64-bit version of Windows 2000 should be able to effortlessly support up to 64 CPUs; beyond that requires actually changing the code substantially (to use multiple words for the bitmaps).

11.4.2 Job, Process, Thread and Fiber Management API Calls

New processes are created using the Win32 API function CreateProcess. This function has 10 parameters, each of which has many options. This design is clearly a lot more complicated than the UNIX scheme, in which fork has no parameters, and exec has just three: pointers to the name of the file to execute, the (parsed) command line parameter array, and the environment strings. Roughly speaking, the 10 parameters to CreateProcess are as follows:

1. A pointer to the name of the executable file.
2. The command line itself (unparsed).
3. A pointer to a security descriptor for the process.
4. A pointer to a security descriptor for the initial thread.
5. A bit telling whether the new process inherits the creator's handles.
6. Miscellaneous flags (e.g., error mode, priority, debugging, consoles).
7. A pointer to the environment strings.

8. A pointer to the name of the new process' current working directory.
9. A pointer to a structure describing the initial window on the screen.
10. A pointer to a structure that returns 18 values to the caller.

Windows 2000 does not enforce any kind of parent-child or other hierarchy. All processes are created equal (no processes are created more equal). However, since 1 of the 18 parameters returned to the creating process is a handle to the new process (allowing considerable control over the new process), there is an implicit hierarchy in terms of who has a handle to whom. Although these handles cannot just be passed directly to other processes, there is a way for a process to make a duplicate handle suitable for another process and then give it the handle, so the implicit process hierarchy may not last long.

Each process in Windows 2000 is created with a single thread, but a process can create more threads later on. Thread creation is simpler than process creation: CreateThread has only six parameters instead of 10:

1. The optional security descriptor.
2. The initial stack size.
3. The starting address.
4. A user-defined parameter.
5. The initial state of the thread (ready or blocked).
6. The thread's ID.

The kernel does the thread creation, so it is clearly aware of threads (i.e., they are not implemented purely in user space as is the case in some other systems).

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; regular pipes cannot.

Mailslots are a feature of Windows 2000 not present in UNIX. They are similar to pipes in some ways, but not all. For one thing, they are one-way, whereas pipes are two-way. They can also 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.

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.

Remote procedure calls 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.

Finally, processes can share memory by mapping onto the same file 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.

Just as Windows 2000 provides numerous interprocess communication mechanisms, it also provides numerous synchronization mechanisms, including semaphores, mutexes, critical regions, and events. All of these mechanisms work on 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 is created using the CreateSemaphore API function, which can initialize it to a given value and define a maximum value as well. Semaphores are kernel 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. Calls for up and down are present, although they have the somewhat peculiar 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).

Mutexes are also kernel 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.

The third synchronization mechanism is based on **critical sections**, (which we have called critical regions elsewhere in this book) which are similar to mutexes, except local to the address space of the creating thread. Because critical sections are not kernel objects, they do not have handles or security descriptors and cannot be passed between processes. Locking and unlocking is done with EnterCriticalSection and LeaveCriticalSection, respectively. Because these API functions are performed initially in user space and only make kernel calls when blocking is needed, they are faster than mutexes.

The last synchronization mechanism uses kernel objects called **events**, of which there are two kinds: **manual-reset events** and **auto-reset events**. Any event can be in one of two states: set and cleared. A thread can wait for an event

to occur with WaitForSingleObject. If another thread signals an event with SetEvent, what happens depends on the type of event. With a manual-reset event, all waiting threads are released and the event stays set until manually cleared with ResetEvent. With an auto-reset 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 set state so a subsequent thread waiting on it is released immediately.

Events, mutexes, and semaphores can all be named and stored in the file system, like named pipes. Two or more processes can synchronize by opening the same event, mutex, or semaphore, rather than having one of them create the object and then make duplicate handles for the others, although the latter approach is certainly an option as well.

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. A summary of the ones discussed above as well as some other important ones are given in Fig. 11-17.

Most of the calls in Fig. 11-17 were either discussed above or should be self-explanatory. Again note that not all of these are system calls. As we mentioned earlier, Windows 2000 knows nothing about fibers. They are entirely implemented in user space. As a consequence, the CreateFiber call does its work entirely in user space without making any system calls (unless it has to allocate some memory). Many other Win32 calls have this property as well, including EnterCriticalSection and LeaveCriticalSection as we noted above.

11.4.3 Implementation of Processes and Threads

Processes and threads are more important and more elaborate than jobs and fibers, so we will concentrate on them here. A process is created when another process makes the Win32 CreateProcess call. This call invokes a (user-mode) procedure in *kernel32.dll* that creates the process in several steps using multiple system calls and other work.

1. The executable file given as a parameter is examined and opened. If it is a valid POSIX, OS/2, 16-bit Windows, or MS-DOS file, a special environment is set up for it. If it is a valid 32-bit Win32 *.exe* file, the registry is checked to see if it is special in some way (e.g., to be run under supervision of a debugger). All of this is done in user mode inside *kernel32.dll*.

2. A system call, NtCreateProcess, is made to create the empty process object and enter it into the object manager's name space. Both the kernel object and the executive object are created. In addition, the

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
SetPriorityClass	Set the priority class for a process
SetThreadPriority	Set the priority for one thread
CreateSemaphore	Create a new semapahore
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

Figure 11-17. Some of the Win32 calls for managing processes, threads, and fibers.

process manager creates a process control block for the object and initializes it with the process ID, quotas, access token, and various other fields. A section object is also created to keep track of the process' address space.

3. When *kernel32.dll* gets control back, it makes another system call, NtCreateThread, to create the initial thread. The thread's user and kernel stacks are also created. The stack size is given in the header of the executable file.

4. *Kernel32.dll* now sends a message to the Win32 environment subsystem telling it about the new process and passing it the process and thread handles. The process and threads are entered into the subsystems tables so it has a complete list of all 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).

5. At this point, the thread is able to run. It starts out by running a run-time system procedure to complete the initialization.
6. The run-time procedure sets the thread's priority, tells loaded DLLs that a new thread is present, and does other housekeeping chores. Finally, it begins running the code of the process' main program.

Thread creation also consists of a number of steps, but we will not go into them in much detail. It starts when the running process executes CreateThread, which calls a procedure inside *kernel32.dll*. This procedure allocates a user stack within the calling process and then makes the NtCreateThread call to create an executive thread object, initialize it, and also create and initialize a thread control block. Again, the Win32 subsystem is notified and enters the new thread in its tables. Then the thread starts running and completes its own initialization.

When a process or thread is created, a handle is returned for it. This handle can be used to start, stop, kill, and inspect the process or thread. It is possible for the owner of a handle to pass the handle to another process in a controlled and secure way. This technique is used to allow debuggers to have control over the processes they are debugging.

Scheduling

Windows 2000 does not have a central scheduling thread. Instead, when a thread cannot run any more, the thread enters kernel mode and runs the scheduler itself to see which thread to switch to. The following conditions cause the currently running thread to execute the scheduler code:

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

In case 1, the thread is already running in kernel mode to carry out the operation on the dispatcher or I/O object. It cannot possibly continue, so it must save its own context, run the scheduler code to pick its successor, and load that thread's context to start 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 run the scheduler to see if the result of its action has released a higher priority thread that is now free to run. If so, a thread switch occurs because Windows 2000 is fully preemptive (i.e., thread switches can occur at any moment, not just at the end of the current thread's quantum).

In case 3, a trap to kernel mode occurs, at which time 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 nothing more important is available, the DPC will run next.

Now we come to the actual scheduling algorithm. The Win32 API provides two hooks for processes to influence thread scheduling. These hooks largely determine the algorithm. First, there is a call SetPriorityClass that sets the priority class of all the threads in the caller's process. The allowed values are: realtime, high, above normal, normal, below normal, and idle.

Second, there is a call SetThreadPriority that sets the relative priority of some thread (possibly, but not necessarily, the calling thread) compared to the other threads in its process. The allowed values are: time critical, highest, above normal, normal, below normal, lowest, and idle. With six process classes and seven thread classes, a thread can have any one of 42 combinations. This is the input to the scheduling algorithm.

The scheduler works as follows. The system has 32 priorities, numbered from 0 to 31. The 42 combinations are mapped onto the 32 priority classes according to the table of Fig. 11-18. 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.

To use these priorities for scheduling, the system maintains an array with 32 entries, corresponding to priorities 0 through 31 derived from the table of Fig. 11-18. Each array entry points to the head of a list of 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 slot 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

		Win32 process class priorities					
		Realtime	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 11-18. Mapping of Win32 priorities to Windows 2000 priorities.

threads ready at the highest priority level, they run round robin for one quantum each. If no thread is ready, the idle thread is run.

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 even know which thread belongs to which process. On a multiprocessor, each CPU schedules itself using the priority array. A spin lock is used to make sure that only one CPU at a time is inspecting the array.

The array of queue headers is shown in Fig. 11-19. The figure shows that there are actually four categories of priorities: realtime, user, zero, and idle, which is effectively −1. These deserve some comment. Priorities 16–31 are called real time, but they are not. There are no guarantees given and no deadlines are met. They are simply higher priority than 0–15. However, priorities 16 through 31 are reserved for the system itself and for threads explicitly assigned those priorities by the system administrator. Ordinary users may not run there for a good reason. If a user thread were to run at a higher priority than, say, the keyboard or mouse thread and get into a loop, the keyboard or mouse thread would never run, effectively hanging the system.

User threads run at priorities 1–15. By setting the process and thread priorities, a user can determine which threads get preference. The zero thread runs in the background and eats up whatever CPU time nobody else wants. Its job is to zero pages for the memory manager. We will discuss its role later. If there is absolutely nothing to do, not even zero pages, the idle thread runs. It is not really a full-blown thread though.

Over the course of time, some patches were made to the basic scheduling algorithm to improve system performance. Under certain specific conditions, the current priority of a user thread can be raised above the base priority (by the operating system), but never above priority 15. Since the array of Fig. 11-19 is

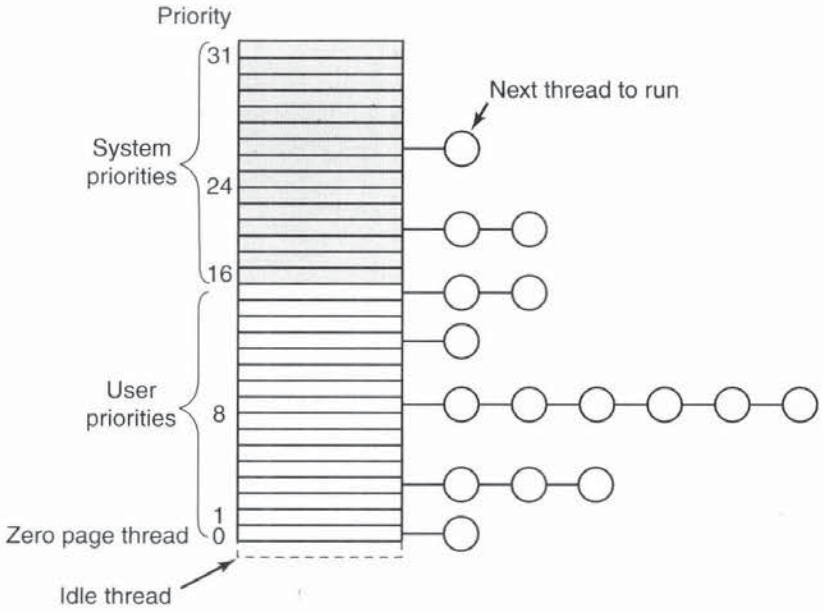


Figure 11-19. Windows 2000 supports 32 priorities for threads.

based on the current priority, changing this priority affects scheduling. No adjustments are ever made to threads running at priority 15 or higher.

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 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 units if it is in the foreground process (the process controlling the window to which keyboard input is sent) and 1 unit 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, but if a thread uses all of its next quantum, it loses one point 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. Clearly, if a thread wants good service, it should play a lot of music.

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. 11-20(a).

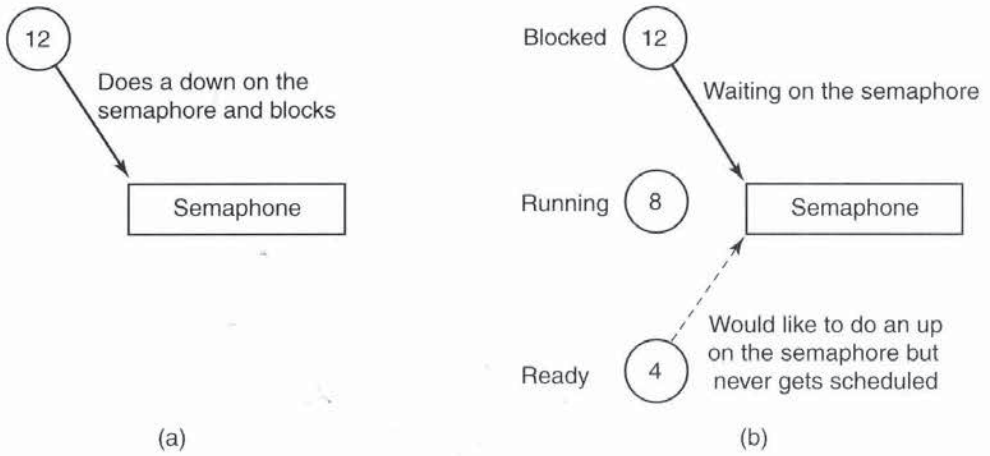


Figure 11-20. An example of priority inversion.

Before the consumer gets a chance to run again, an unrelated thread at priority 8 becomes ready and starts running, as shown in Fig. 11-20(b). As long as this thread wants to run, it will be able to, since it outguns the consumer and the producer, while higher, is blocked. Under these circumstances, the producer will never get to run again until the priority 8 thread gives up.

Windows 2000 solves this problem through what might be charitably called a big hack. The system keeps track of how long it has been since a ready thread ran last. If it exceeds a certain threshold, it is moved to priority 15 for two quanta. This may give it the opportunity to unblock the producer. After the two quanta are up, the boost is abruptly removed rather than decaying gradually. Probably a better solution would be to penalize threads that use up their quantum over and over by lowering their priority. After all, the problem was not caused by the starved thread, but by the greedy thread. This problem is well known under the name **priority inversion**.

An analogous problem happens if a priority 16 thread grabs a mutex and does not get a chance to run for a long time, starving more important system threads that are waiting for the mutex. This problem can be prevented within the operating system by having a thread that needs a mutex for a short time just disable scheduling while it is busy. On a multiprocessor, a spin lock should be used.

Before leaving the subject of scheduling, it is worth saying a couple of words about the quantum. On Windows 2000 Professional the default is 20 msec; on uniprocessor servers it is 120 msec; on multiprocessors various other values are used, depending on the clock frequency. The shorter quantum favors interactive users whereas the longer quantum reduces context switches and thus provides better efficiency. This is the meaning of the last column in Fig. 11-2. These defaults can be increased manually by 2x, 4x, or 6x if desired. As an aside, the size of the quantum was chosen a decade ago and not changed since although machines are now more than an order of magnitude faster. The numbers probably

could be reduced by a factor of 5 to 10 with no harm and possibly better response time for interactive threads in a heavily loaded system.

One last patch to the scheduling algorithm says that when a new window becomes the foreground window, all of its threads get a longer quantum by an amount taken from the registry. This change gives them more CPU time, which usually translates to better service for the window that just moved to the foreground.

11.4.4 MS-DOS Emulation

One of the design goals of Windows 2000 was inherited from NT: try to run as many reasonable MS-DOS programs as possible. This goal is quite different from Windows 98's stated goal: run all old MS-DOS programs (to which we add: no matter how ill-behaved they may be).

The way Windows 2000 deals with ancient programs is to run them in a fully protected environment. When an MS-DOS program is started, a normal Win32 process is started and loaded with an MS-DOS emulation program, *ntvdm* (NT Virtual DOS Machine) that will monitor the MS-DOS program and carry out its system calls. Since MS-DOS only recognized memory up to 1 MB on the 8088 and only up to 16 MB with bank switching and other tricks on the 286, it is safe to put *ntvdm* high in the process' virtual address space where the program has no way to address it. This situation is shown in Fig. 11-21.

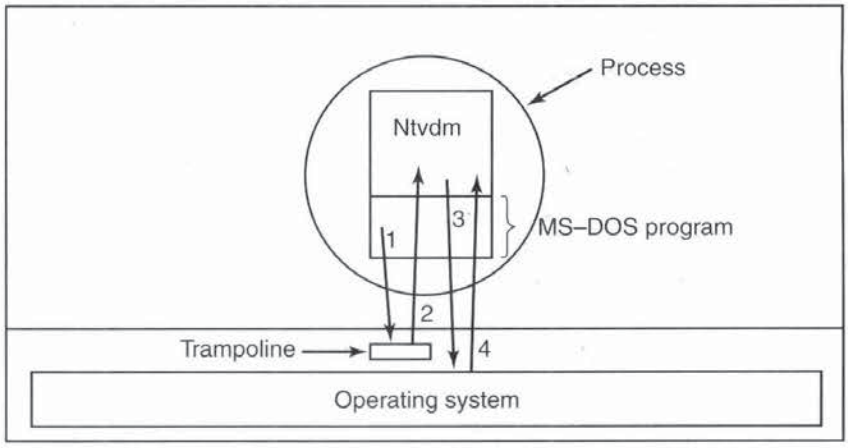


Figure 11-21. How old MS-DOS programs are run under Windows 2000.

When the MS-DOS program is just executing normal instructions, it can run on the bare hardware since the Pentium includes all the 8088 and 286 instructions as subsets. The interesting part is what happens when the MS-DOS program wants to do I/O or interact with the system. A well-behaved program just makes a system call. In expectation of this, *ntvdm* instructs Windows 2000 to reflect all MS-DOS system calls back to it. In effect, the system call just bounces off the operating

system and is caught by the emulator, as shown in steps 1 and 2 in Fig. 11-21. Sometimes this technique is referred to as using a **trampoline**.

Once it gets control, the emulator figures out what the program was trying to do and issues its own Win32 calls to get the work done (step 3 and 4 in Fig. 11-21). As long as the program is well behaved and just makes legal MS-DOS system calls, this technique works fine. The trouble is that some old MS-DOS programs bypassed the operating system and wrote directly to the video RAM, read directly from the keyboard, and so on, things that are impossible in a protected environment. To the extent that the illegal behavior causes a trap, there is some hope that the emulator can figure out what the program was trying to do and emulate it. If it does not know what the program wants, the program is just killed because 100 percent emulation was not a Windows 2000 requirement.

11.4.5 Booting Windows 2000

Before Windows 2000 can start up, it must be booted. The boot process creates the initial processes that bring up the system. In this section we will briefly discuss how the boot process works for Windows 2000. The hardware boot process consists of reading in the first sector of the first disk (the master boot record) and jumping to it, as we described in Sec. 5.4.2. This short assembly language program reads the partition table to see which partition contains the bootable operating system. When it finds the operating system partition, it reads in the first sector of that partition, called the **boot sector**, and jumps to it. The program in the boot sector reads its partition's root directory, searching for a file called *ntldr* (another piece of archaeological evidence that Windows 2000 is really NT). If it finds that file, it reads the file into memory and executes it. *Ntldr* loads Windows 2000. As an aside, there are several versions of the boot sector, depending on whether the partition is formatted as FAT-16, FAT-32, or NTFS. When Windows 2000 is installed, the correct version of the master boot record and boot sector are written to disk.

Ntldr now reads a file called *Boot.ini*, which is the only configuration information not in the registry. It lists all the versions of *hal.dll* and *ntoskrnl.exe* available for booting in this partition. The file also provides many parameters, such as how many CPUs and how much RAM to use, whether to give user processes 2 GB or 3 GB, and what rate to set the real-time clock to. *Ntldr* then selects and loads *hal.dll* and *ntoskrnl.exe* files as well as *bootvid.dll*, the default video driver for writing on the display during the boot process. *Ntldr* next reads the registry to find out which drivers are needed to complete the boot (e.g., the keyboard and mouse drivers, but also dozens more for controlling various chips on the parentboard). Finally, it reads in all these drivers and passes control to *ntoskrnl.exe*.

Once started, the operating system does some general initialization and then calls the executive components to do their own initialization. For example, the object manager prepares its name space to allow other components call it to insert

their objects into the name space. Many components also do specific things related to their function, such as the memory manager setting up the initial page tables and the plug-and-play manager finding out which I/O devices are present and loading their drivers. All in all, dozens of steps are involved, during which time the progress bar displayed on the screen is growing in length as steps are completed. The last step is creating the first true user process, the **session manager**, *smss.exe*. Once this process is up and running, booting is completed.

The session manager is a native Windows 2000 process. It makes true system calls and does not use the Win32 environment subsystem, which is not even running yet. In fact, one of its first duties is to start it (*csrss.exe*). It also reads the registry hives from disk and learns what else it is supposed to do. Typically its work includes entering many objects in the object manager's name space, creating any extra paging files needed, and opening important DLLs to have them around all the time. After it has done most of this work, it creates the login daemon, *winlogon.exe*.

At this point, the operating system is up and running. Now it is time to get the service processes (user space daemons) going and allow users to log in. *Winlogon.exe* first creates the authentication manager (*lsass.exe*), and then the parent process of all the services (*services.exe*). The latter looks in the registry to find out which user space daemon processes are needed and what files they are in. It then starts creating them. They are shown in Fig. 11-7. The fact that the disk is generally being heavily used after the first user has logged in (but has done nothing) is not the user's fault. The culprit is *services.exe* creating all the services. In addition, it also loads any remaining device drivers that have not yet been loaded. The hierarchy of initial processes and some typical services are shown in Fig. 11-22.

Winlogon.exe is also responsible for all user logins. The actual login dialog is handled by a separate program in *msgina.dll* to make it possible for third parties to replace the standard login with faceprint identification or something else other than name and password. After a successful login, *winlogon.exe* gets the user's profile from the registry and from it determines which shell to run. Many people do not realize it, but the standard Windows desktop is just *explorer.exe* with some options set. If desired, a user can select any other program as the shell, including the command prompt or even *Word*, by editing the registry. However, editing the registry is not for the faint of heart; a mistake here can make the system unusable.

11.5 MEMORY MANAGEMENT

Windows 2000 has an extremely sophisticated virtual memory system. It has a number of Win32 functions for using it and part of the executive plus six dedicated kernel threads for managing it. In the following sections we will look at the fundamental concepts, the Win32 API calls, and finally the implementation.

Process	Description
idle	Not really a process, but home to the idle thread
system	Creates smss.exe & paging files; reads registry; opens DLLs
smss.exe	First real proc; much initialization; creates csrss & winlogon
csrss.exe	Win32 subsystem process
winlogon.exe	Login daemon
lsass.exe	Authentication manager
services.exe	Looks in registry and starts services
Printer server	Allows remote jobs to use the printer
File server	Serves requests for local files
Telnet daemon	Allows remote logins
Incoming email handler	Accepts and stores inbound email
Incoming fax handler	Accepts and prints inbound faxes
DNS resolver	Internet domain name system server
Event logger	Logs various system events
Plug-and-play manager	Monitors hardware to see what is out there

Figure 11-22. The processes starting up during the boot phase. The ones above the line are always started. The ones below it are examples of services that could be started.

11.5.1 Fundamental Concepts

In Windows 2000, every user process has its own virtual address space. Virtual addresses are 32 bits long, so each process has 4 GB of virtual address space. The lower 2 GB minus about 256 MB are available for the process' code and data; the upper 2 GB map onto kernel memory in a protected way. The virtual address space is demand paged, with a fixed page size (4 KB on the Pentium).

The virtual address space layout for three user processes is shown in Fig. 11-23 in slightly 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. Invalid pointers are often 0 or −1, so attempts to use them on Windows 2000 will cause an immediate trap instead of reading garbage or, worse yet, writing to an incorrect memory location. However, when old MS-DOS programs are being run in emulation mode, they can be mapped in.

Starting at 64 KB comes the user's private code and data. This extends up to almost 2 GB. The last piece of the bottom 2 GB contains some system counters and timers that are shared among all users read only. Making them visible here allows processes to access them without the overhead of a system call.

The upper 2 GB contains the operating system, including the code, data, and the paged and nonpaged pools (used for objects, etc.). The upper 2 GB is shared among all user processes, except for the page tables, which are each process' own page tables. The upper 2 GB of memory is not writable and mostly not even

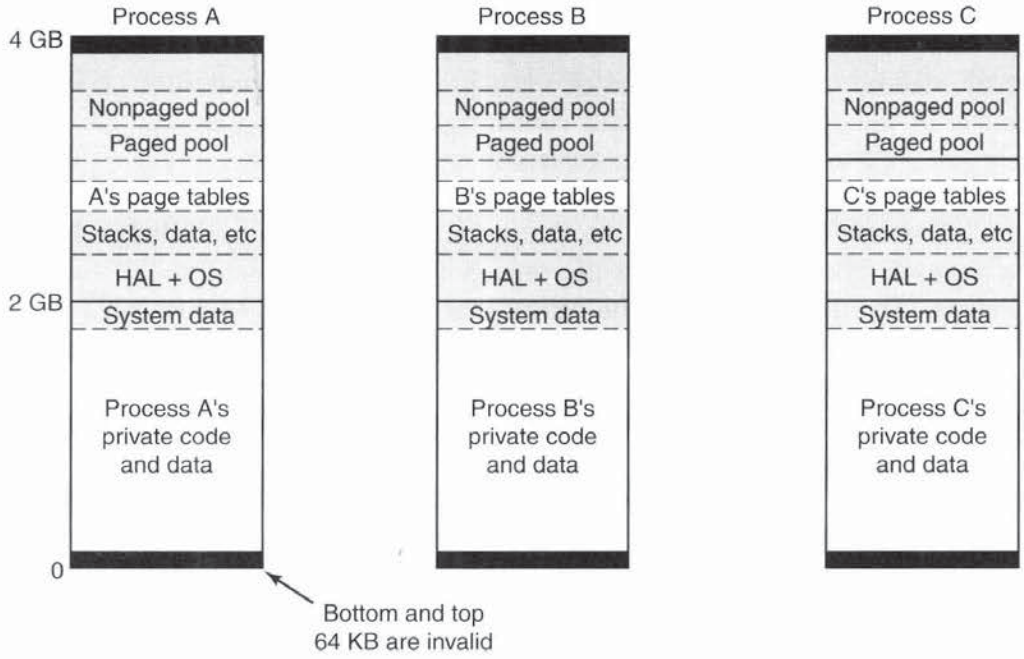


Figure 11-23. Virtual address space layout for three user processes. The white areas are private per process. The shaded areas are shared among all processes.

readable for user-mode processes. The reason for putting it here is that when a thread makes a system call, it traps into kernel mode and just keeps on running in the same thread. By making the whole operating system and all of its data structures (as well as the whole user process) visible within a thread's address space when it enters kernel mode, there is no need to change the memory map or flush the cache upon kernel entry. All that has to be done is switch to the thread's kernel stack. The trade-off here is less private address space per process in return for faster system calls. Large database servers already feel cramped, which is why the 3-GB user space option is available on Advanced Server and Datacenter Server.

Each virtual page can be in one of three states: free, reserved, or committed. A **free page** is not currently in use and a reference to it causes a page fault. When a process is started, all of its pages are in free state until the program and initial data are mapped into its address space. Once code or data is mapped onto a page, the page is said to be **committed**. A reference to a committed page is mapped using the virtual memory hardware and succeeds if the page is in main memory. If the page is not in main memory, a page fault occurs and the operating system finds and brings in the page from disk.

A virtual page can also be in **reserved** state, meaning it is not available for being mapped until the reservation is explicitly removed. For example, when a

new thread is created, 1 MB of stack space is reserved in the process' virtual address space, but only one page is committed. This technique means that the stack can eventually grow to 1 MB without fear that some other thread will allocate the needed contiguous piece of virtual address space out from under it. In addition to the free, reserved, and committed attributes, pages also have other attributes, such as being readable, writable, and executable.

An interesting trade-off occurs with assignment of backing store to committed pages. A simple strategy would be to assign a page in one of the paging files to back up each committed page at the time the 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. The downside of this strategy is that the paging file might have to be as large as the union of all processes' virtual memory. On a large system that rarely ran out of memory and thus rarely paged, this approach would waste disk space.

To avoid wasting disk space, Windows 2000 committed pages that have no natural home on the disk (e.g., stack pages) are not assigned a disk page until the moment that they have to be paged out. This design makes the system more complex because the paging files maps may have to be fetched during a page fault, and fetching them may cause one or more additional page faults inside the page fault handler. On the other hand, no disk space need be allocated for pages that are never paged out.

Trade-offs like this (system complexity versus better performance or more features) tend to get resolved in favor of the latter because the value of better performance or more features is clear but the downside of complexity (a bigger maintenance headache and more crashes per year) is hard to quantify. Free and reserved pages never have shadow pages on disk and references to them always cause page faults.

The shadow pages on the disk are arranged into one or more paging files. There may be up to 16 paging files, possibly spread over 16 separate disks, for higher I/O bandwidth. Each one has an initial size and a maximum size it can grow to later if needed. These files can be created at the maximum size at system installation time in order to reduce the chances that they are highly fragmented, but new ones can be created using the control panel later on. The operating system keeps track of which virtual page maps onto which part of which paging file. For (execute only) program text, the executable binary file (i.e., *.exe* or *.dll* file) contains the shadow pages; for data pages, the paging files are used.

Windows 2000, like many versions of UNIX, allows files to be mapped directly onto regions of the virtual address spaces (i.e., runs of consecutive pages). Once a file has been mapped onto the address space, it can be read or written using ordinary memory references. Memory-mapped files are implemented in the same way as other committed pages, only the shadow pages are in the user's file instead of in the paging file. As a result, while a file is mapped in, the version in memory may not be identical to the disk version (due to recent writes to the vir-

tual address space). However, when the file is unmapped or is explicitly flushed, the disk version is brought up to date.

Windows 2000 explicitly allows two or more processes to map onto the same part of the same file at the same time, possibly at different virtual addresses, as shown in Fig. 11-24. By reading and writing memory words, the processes can now communicate with each other and pass data back and forth at very high bandwidth, since no copying is required. Different processes may have different access permissions. Since all the processes using a mapped file share the same pages, changes made by one of them are immediately visible to all the others, even if the disk file has not yet been updated. Care is also taken that if another process opens the file for normal reading, it sees the current pages in RAM, not stale pages from the disk.

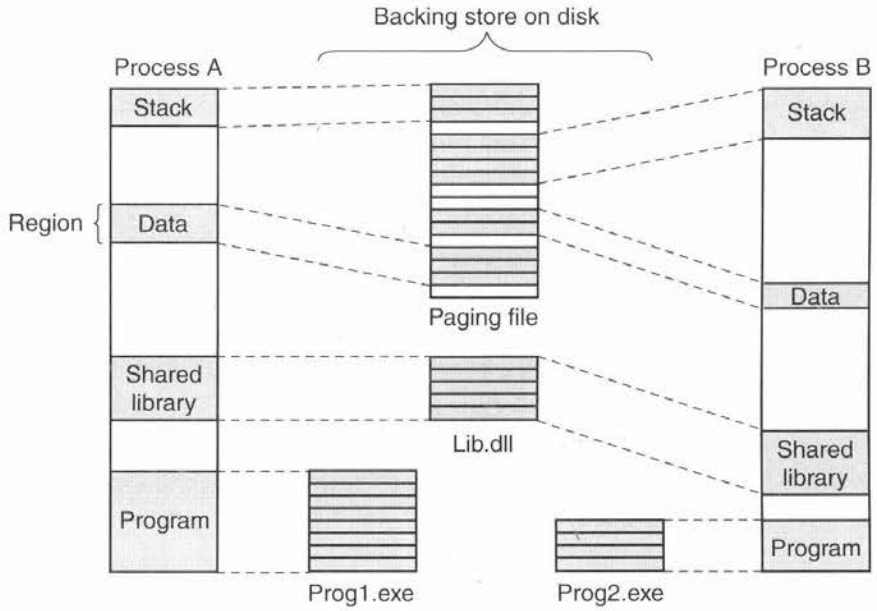


Figure 11-24. Mapped regions with their shadow pages on disk. The *lib.dll* file is mapped into two address spaces at the same time.

It is worth noting that there is a problem if two programs share a DLL file and one of them changes the file's static data. If no special action is taken, the other one will see the changed data, which is probably not what is desired. The problem is solved by mapping all pages in as read only by secretly noting that some are really writable. When a write happens to a page that is mapped read only but is really writable, a private copy of the page is made and mapped in. Now it can be written safely without affecting other users or the original copy on disk. This technique is called **copy-on-write**.

Also it is worth noting that if program text is mapped into two address spaces at different addresses, a certain problem arises with addressing. What happens if the first instruction is JMP 300? If process one maps the program in at address

65,536, the code can easily be patched to read JMP 65836. But what happens if a second process now maps it in at 131,072? The JMP 65836 will go to address 65,836 instead of 131,372 and the program will fail. The solution is to use only relative offsets, not absolute virtual addresses in code that is to be shared. Fortunately, most machines have instructions using relative offsets as well as instructions using absolute addresses. Compilers can use the relative offset instructions, but they have to know in advance whether to use them or the absolute ones. The relative ones are not used all the time because the resulting code is usually less efficient. Usually, a compiler flag tells them which to use. The technique of making it possible to place a piece of code at any virtual address without relocation is called **position independent code**.

Years ago, when 16-bit (or 20-bit) virtual address spaces were standard, but machines had megabytes of physical memory, all kinds of tricks were thought of to allow programs to use more physical memory than fit in the address space. Often these tricks went under the name of **bank switching**, in which a program could substitute some block of memory above the 16-bit or 20-bit limit for a block of its own memory. When 32-bit machines were introduced, people thought they would have enough address space forever. They were wrong. The problem is back. Large programs often need more than the 2 GB or 3 GB of user address space Windows 2000 allocates to them, so bank switching is back, now called **address windowing extensions**. This facility allows programs to map into shuffle chunks of memory in and out of the user portion of the address space (and especially above the dreaded 4-GB boundary). Since it is only used on servers with more than 2 GB of physical memory, we will defer the discussion until the next edition of this book (by which time even entry-level desktop machines will be feeling the 32-bit pinch).

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. 11-25. All of them operate on a region consisting either of a single page or a sequence of two or more pages that are consecutive in the virtual address space.

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 (up to 64 KB). 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 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 this ability, for example. A limit is enforced by the operating system to prevent processes from getting too greedy. Actually, the pages can be removed from memory, but only if the entire

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 11-25. The principal Win32 API functions for managing virtual memory in Windows 2000.

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. 11-25, Windows 2000 also has API functions to allow a process to access the virtual memory of a different process over which it has been given control (i.e., for which it has a handle).

The last four API functions listed are for managing memory-mapped files. To map a file, a file mapping object (see Fig. 11-10) must first be created, with CreateFileMapping. This function returns a handle to the file mapping object and optionally enters a name for it into the file system so another process can use it. The next two functions map and unmap files, respectively. The last one can be used by a process to map in a file currently also mapped in by a different process. 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 memory.

11.5.3 Implementation of Memory Management

Windows 2000 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 two up to 64 KB. On the Pentium they are fixed at 4 KB; on the Itanium they can be 8 KB or 16 KB. In addition, the operating system itself can use 4-MB pages to reduce page table space consumed.

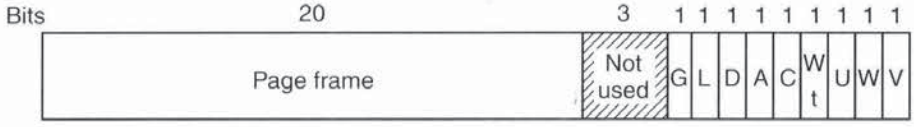
Unlike the scheduler, which selects individual threads to run and does not care much about processes, the memory manager deals entirely with processes and does not care much about threads. After all, processes, not threads, own the address space and that is what the memory manager deals with. When a region of virtual address space is allocated, as four of them have been for process *A* in

Fig. 11-24, the memory manager creates a **VAD (Virtual Address Descriptor)** for it, listing the range of addresses mapped, the backing store file and offset where it is mapped, and the protection code. When the first page is touched, the directory of page tables is created and a pointer to it inserted is in the VAD. In fact, an address space is completely defined by the list of its VADs. This scheme supports sparse address spaces because unused areas between the mapped regions use no resources.

Page Fault Handling

Windows 2000 does not use any form of prepaging. When a process starts, none of its pages are in memory. All of them are brought in dynamically as page faults occurs. On each page fault, a trap to the kernel (in the sense of Fig. 11-7) occurs. The kernel builds a machine-independent descriptor telling what happened and passes this to the memory manager part of the executive. The memory manager then checks it for validity. If the faulted page falls within a committed or reserved region, it looks up the address in the list of VADs, finds (or creates) the page table, and looks up the relevant entry.

The page table entries are different for different architectures. For the Pentium, the entry for a mapped page is shown in Fig. 11-26. Unmapped pages also have entries, but their format is somewhat different. For example, for an unmapped page that must be zeroed before it may be used, that fact is noted in the page table.



G: Page is global to all processes
L: Large (4-MB) page
D: Page is dirty
A: Page has been accessed
C: Caching enabled/disabled
Wt: Write through (no caching)
U: Page is accessible in user mode
W: Writing to the page permitted
V: Valid page table entry

Figure 11-26. A page table entry for a mapped page on the Pentium.

The most important bits in the page table entry for purposes of the paging algorithm are the *A* and *D* bits. They are fed by the hardware and keep track of whether the page has been referenced or written on, respectively, since the last time they were cleared.

Page faults come in five categories:

1. The page referenced is not committed.
2. A protection violation occurred.
3. A shared page has been written.

4. The stack needs to grow.
5. The page referenced is committed but not currently mapped in.

The first and second cases are fatal errors from which there is no recovery for the faulting 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. The solution is to copy the page to a new physical page frame and map that one in read/write. This is how copy-on-write works. (If a shared page is marked writable in all processes using it, it is not copy-on-write and no fault occurs when writing to it.) The fourth case requires allocating a new page frame and mapping it in. However, the security rules require that the page contain only 0s, to prevent the process from snooping on the previous owner of the page. Thus a page of 0s must be found, or if one is not available, another page frame must be allocated and zeroed on the spot. Finally, the fifth case is a normal page fault. The page is located and mapped in.

The actual mechanics of getting and mapping pages is fairly standard, so we will not discuss this issue. The only noteworthy feature is that Windows 2000 does not read in isolated pages from the disk. Instead, it reads in runs of consecutive pages, usually about 1–8 pages, in an attempt to minimize the number of disk transfers. The run size is larger for code pages than for data pages.

The Page Replacement Algorithm

Page replacement works like this. The system makes a serious attempt to maintain a substantial number of free pages in memory so that when a page fault occurs, a free page can be claimed on the spot, without the need to first write some other page to disk. As a consequence of this strategy, most page faults can be satisfied with at most one disk operation (reading in the page), rather than sometimes two (writing back a dirty page and then reading in the needed page).

Of course, the pages on the free list have to come from somewhere, so the real page replacement algorithm is how pages get taken away from processes and put on the free list (actually, there are four free lists, but for the moment it is simplest to think of there being just one; we will come to the details later). Let us now take a look at how Windows 2000 frees pages. To start with, the entire paging system makes 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 can be thus 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. The default initial minimum is in the range 20–50 and the default initial maximum is in the range 45–345, depending on the total amount of RAM. The system administrator can change these defaults, however.

If a page fault occurs and the working set is smaller than the minimum, the page is added. On the other hand, if a page fault occurs and the working set is larger than the maximum, a page is evicted from the working set (but not from memory) to make room for the new page. This algorithm means that Windows 2000 uses a local algorithm, to prevent one process from hurting others by hogging memory. However, the system does try to tune itself to some extent. For example, if it observes that one process is paging like crazy (and the others are not), the system may increase the size of its maximum working set, so that over time, the algorithm is a mix of local and global. There is an absolute limit on the working set size, however: even if there is only one process running, it may not take the last 512 pages, to leave some slack for new processes.

So far, so good, but the story is not over yet. Once a second, a dedicated kernel daemon thread, the **balance set manager**, checks to see if there are enough free pages. If there are not enough, it starts the **working set manager** thread to examine the working sets and recover more pages. The working set manager first determines the order to examine the processes in. Large processes that have been idle for a long time are considered before small active processes and the foreground process is considered last.

The working set manager then starts inspecting processes in the chosen order. If a process' working set is currently less than its minimum or it has incurred more than a certain number of page faults since the last inspection, it is passed over. Otherwise, one or more pages are removed. The target number of pages to remove is a complicated function of the total RAM size, how tight memory is, how the current working set size compares to the process' minimum and maximum, and other parameters. All the pages are examined in turn.

On a uniprocessor, if a page's reference bit is clear, a counter associated with the page is incremented. If the reference bit is set, the counter is set to zero. After the scan, the pages with the highest counters are removed from the working set. The thread continues examining processes until it has recovered enough pages, then it stops. If a complete pass through all processes still has not recovered enough pages, it makes another pass, trimming more aggressively, even reducing working sets below their minimum if necessary.

On a multiprocessor, looking at the reference bit does not work because although the current CPU may not have touched the page recently, some other one may have. Examining another CPU's reference bits is too expensive to do. Consequently, the reference bit is not examined and the oldest pages are removed.

It should be noted that for page replacement purposes, the operating system itself is regarded as a process. It owns pages and also has a working set. This working set can be trimmed. However, parts of the code and the nonpaged pool are locked in memory and cannot be paged out under any circumstances.

Physical Memory Management

Above we mentioned that there were actually four free lists. Now it is time to see what all of them are for. Every page in memory is either in one or more working sets or on exactly one of these four lists, which are illustrated in Fig. 11-27. The standby (clean) and modified (dirty) lists hold pages that have recently been evicted from a working set, are still in memory, and are still associated with the process that was using them. The difference between them is that clean pages have a valid copy on disk and can thus be abandoned at will, whereas dirty pages do not have an up-to-date copy on disk. The free list consists of clean pages that are no longer associated with any process. The pages on the zeroed page list are not associated with any process and are also filled with zeros. A fifth list holds any physically defective RAM pages that may exist to make sure that they are not used for anything.

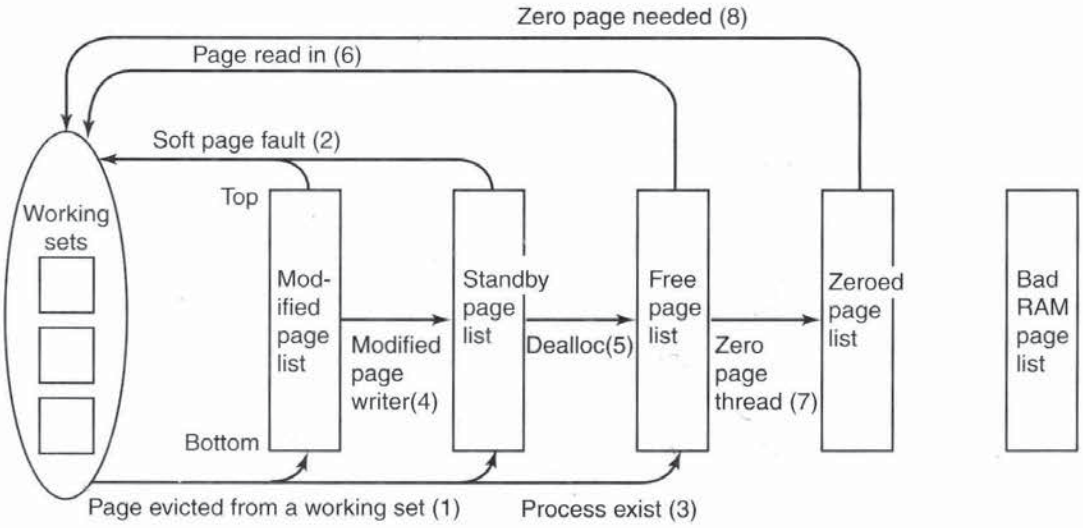


Figure 11-27. The various page lists and the transitions between them.

Pages are moved between the working sets and the various lists by the working set manager and other kernel daemon 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). 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 faulted back to it, so they go on the free list (3). These pages are no longer associated with any process.

Other transitions are caused by other daemon threads. Every 4 seconds the **swapper 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 and their pages are moved to the standby or modified lists, also shown as (1).

Two other daemon threads, the **mapped page writer** and the **modified page writer**, wake up periodically to see if there are enough clean pages. If there are 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 paging files. The result of these writes is to transform dirty pages into 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, which never grows. Nobody ever said Windows 2000 was simple.

The other transitions in Fig. 11-27 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 daemon thread, the **zero page thread**, runs at the lowest priority (see Fig. 11-19), 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.

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? If the CPU is idle a lot and the zero page thread gets to run often, taking a zeroed page is better because there is no shortage of them. However, if the CPU is always busy and the disk is mostly idle, it is better to take a page from the standby list to avoid the CPU cost of having to zero another page later if a stack grows.

Another puzzle. How aggressively should the daemons move pages from the modified list to the standby list? Having clean pages around is better than having dirty pages around (since they 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.

In general, Windows 2000 resolves these kinds of conflicts through complex heuristics, guesswork, historical precedent, rules of thumb, and administrator-controlled parameter settings. Furthermore, the code is so complex that the designers are loathe to touch parts of it for fear of breaking something somewhere else in the system that nobody really understands any more.

To keep track of all the pages and all the lists, Windows maintains a page frame database with as many entries as there are RAM pages, as shown in Fig. 11-28. This 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., valid versus invalid). Valid entries maintain the page's state and a count of how many page tables point to the page, so the system can tell when the page is no longer in use. Pages that are in a working set tell which one. There is also a pointer to the page table pointing to the page, if any (shared pages are handled specially), 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, etc.

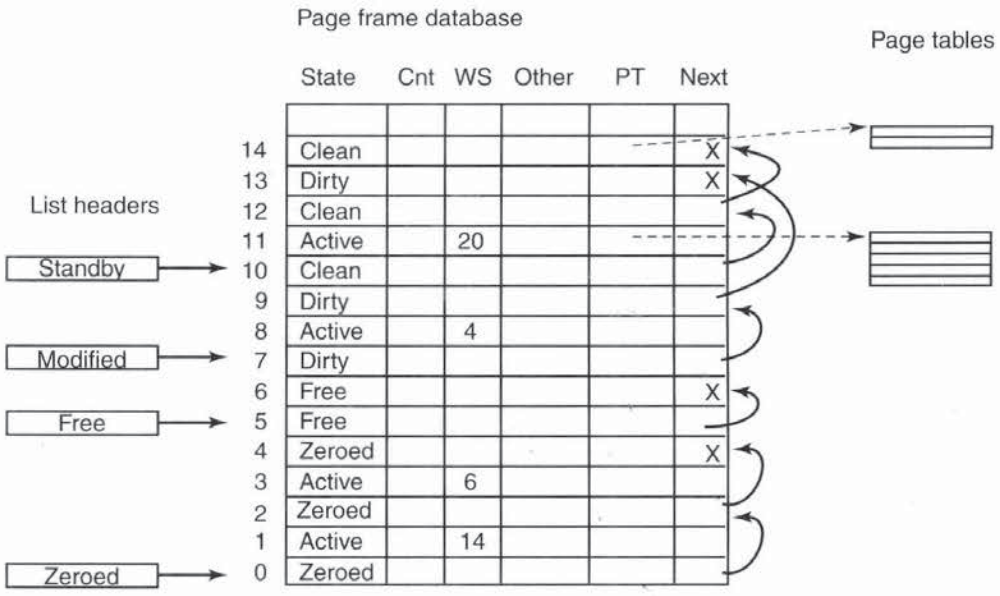


Figure 11-28. Some of the major fields in the page frame database for a valid page.

All in all, memory management is a highly complex subsystem 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 INPUT/OUTPUT IN WINDOWS 2000

The goal of the Windows 2000 I/O system is to provide a framework for efficiently handling a very wide variety of I/O devices. Current input devices include various kinds of keyboards, mice, touch pads, joysticks, scanners, still cameras, television cameras, bar code readers, microphones, and laboratory rats. Current output devices include monitors, printers, plotters, beamers, CD-recorders, and sound cards. Storage devices include floppy disks, IDE and SCSI hard disks, CD-ROMs, DVDs, Zip drives, and tape drives. Finally, other devices include clocks, networks, telephones, and camcorders. No doubt many new I/O devices will be invented in the years to come, so Window 2000 has been designed with a general framework to which new devices can easily be attached. In the following sections we will examine some of the issues relating to I/O.

11.6.1 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, USB, IEEE 1394, and SCSI, 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 one is loaded, a **driver object** is created for it. For some buses, such as SCSI, enumeration happens only at boot time, but for other buses, such as USB and IEEE 1394, it can happen at any moment, requiring close contact between the plug-and-play manager, the bus driver (which actually does the enumeration), and the I/O manager.

The I/O manager is also closely connected with the power manager. The power manager can put the computer into any of six states, roughly described as:

1. Fully operational.
2. Sleep-1: CPU power reduced, RAM and cache on; instant wake-up.
3. Sleep-2: CPU and RAM on; CPU cache off; continue from current PC.
4. Sleep-3: CPU and cache off; RAM on; restart from fixed address.
5. Hibernate: CPU, cache, and RAM off; restart from saved disk file.
6. Off: Everything off; full reboot required.

I/O devices can also be in various power states. Turning them on and off is handled by the power manager and I/O manager together. Note that states 2 through 6 are only used when the CPU has been idle for a shorter or longer time interval.

Somewhat surprisingly, all the file systems are technically I/O drivers. Requests for data blocks from user processes are initially sent to the cache manager. If the cache manager cannot satisfy the request from the cache, it has the I/O manager call the proper file system driver to go get the block it needs from disk.

An interesting feature of Windows 2000 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.

Another interesting aspect of Windows 2000 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.

11.6.2 Input/Output API Calls

Windows 2000 has over 100 separate APIs for a wide variety of I/O devices, including mice, sound cards, telephones, tape drives, etc. Probably the most important is the graphics system, for which there are thousands of Win32 API calls. In Sec. 5.7.3 we began our discussion of the Window graphical system. Here we will continue, mentioning a few of the Win32 API categories, each of which has many calls. A brief summary of the categories are given in Fig. 11-29. As we mentioned in Chap. 5, multiple 1500-page books have been written on the graphics portion of the Win32 API.

Win32 calls exist to create, destroy, and manage windows. Windows have a vast number of styles and options that can be specified, including titles, borders, colors, sizes, and scroll bars. Windows can be fixed or movable, of constant size or resizable. Their properties can be queried and messages can be sent to them.

Many windows contain menus, so there are Win32 calls for creating and deleting menus and menu bars. Dynamic menus can be popped up and removed. Menu items can be highlighted, dimmed out, or cascaded.

Dialog boxes are popped up to inform the user of some event or ask a question. They may contain buttons, sliders, or text fields to be filled in. Sounds can also be associated with dialog boxes, for example for warning messages.

There are hundreds of drawing and painting functions available, ranging from setting a single pixel to doing complex region clipping operations. Many calls are provided for drawing lines and closed geometric figures of various kinds, with detailed control over textures, colors, widths, and many other attributes.

API group	Description
Window management	Create, destroy, and manage windows,
Menus	Create, destroy, and append to menus and menu bars
Dialog boxes	Pop up a dialog box and collect information
Painting and drawing	Display points, lines, and geometric figures
Text	Display text in some font, size, and color
Bitmaps and icons	Placement of bitmaps and icons on the screen
Colors and palettes	Manage the set of colors available
The clipboard	Pass information from one application to another
Input	Get information from the mouse and keyboard

Figure 11-29. Some categories of Win32 API calls.

Another group of calls relates to displaying text. Actually, the text display call, TextOut, is straightforward. It is the management of the color, point sizes, typefaces, character widths, glyphs, kerning, and other typesetting details where the complexity comes in. Fortunately, the rasterization of text (conversion to bitmaps) is generally done automatically.

Bitmaps are small rectangular blocks of pixels that can be placed on the screen using the BitBlt Win32 call. They are used for icons and occasionally text. Various calls are provided for creating, destroying, and managing icon objects.

Many displays use a color mode with only 256 or 65,536 of the 2^{24} possible colors in order to represent each pixel with only 1 or 2 bytes, respectively. In these cases a color palette is needed to determine which 256 or 65,536 colors are available. The calls in this group create, destroy, and manage palettes, select the nearest available color to a given color, and try to make colors on the screen match colors on color printers.

Many Windows 2000 programs allow the user to select some data (e.g., a block of text, part of a drawing, a set of cells in a spreadsheet), put it on the clipboard, and allow it to be pasted into another application. The clipboard is generally used for this transmission. Many clipboard formats are defined, including text, bitmaps, objects, and metafiles. The latter are sets of Win32 calls that when executed draw something, allowing arbitrary drawings to be cut and pasted. This group of calls puts things on the clipboard, takes things off the clipboard, and generally manages it.

Finally, we come to input. There are no Win32 calls for GUI applications for reading input from the keyboard because GUI applications are event driven. The main program consists of a big loop getting input messages. When the user types something interesting, a message is sent to the program telling it what just came in. On the other hand, there are calls relating to the mouse, such as reading its (x, y) position and the state of its buttons. Some of the input calls are actually output

calls, though, such as selecting a mouse cursor icon and moving it around the screen (basically, this is output to the screen). For nonGUI applications, it is possible to read from the keyboard.

11.6.3 Implementation of I/O

We could go on more-or-less indefinitely about the Win32 graphics calls, but now it is time to look at how the I/O manager implements graphics and other I/O functions. The main function of the I/O manager is to create a framework in which different I/O devices can operate. The basic structure of the framework is a set of device-independent procedures for certain aspects of I/O plus a set of loaded device drivers for communicating with the devices.

11.6.4 Device Drivers

To make sure that device drivers work well with the rest of Windows 2000, Microsoft has defined a **Windows Driver Model** that device drivers are expected to conform with. Furthermore, it also has provided a tool kit that is designed to help driver writers produce conformant drivers. In this section we will briefly examine this model. Conformant drivers must meet all of the following requirements as well as some others:

1. Handle incoming I/O requests, which arrive in a standard format.
2. Be as object based as the rest of Windows 2000.
3. Allow plug-and-play devices to be dynamically added or removed.
4. Permit power management, where applicable.
5. Be configurable in terms of resource usage.
6. Be reentrant for use on multiprocessors.
7. Be portable across Windows 98 and Windows 2000

I/O Requests are passed to drivers in the form of a standardized packet called an **IRP (I/O Request Packet)**. Conformant drivers must be able to handle them. Drivers must be object based in the sense of supporting a specific list of methods that the rest of the system can call. They must also correctly deal with other Windows 2000 objects when given an object handle to deal with.

Conformant drivers must fully support plug and play, which means that if a device managed by the driver is suddenly added or removed from the system, the driver must be prepared to accept this information and act on it, even in the case that the device currently being accessed is suddenly removed. Power management must also be supported for devices for which this is relevant. For example, if the system decides it is now time to go into a low-power hibernation mode, all

devices that are capable of doing this must do so to save energy. They must also wake up when told to do so.

Drivers must be configurable, which means not having any built-in assumptions about which interrupt lines or I/O ports certain devices use. For example, the printer port on the IBM PC and its successors has been at address 0x378 for more than 20 years and it is unlikely to change now. But a printer driver that has this address hard coded into it is not conformant.

Being multiprocessor safe is also a requirement as Windows 2000 was designed for use on multiprocessors. Concretely, this requirement means while a driver is actively running and processing one request on behalf of one CPU, a second request may come in on behalf of a different CPU. The second CPU may begin executing the driver code simultaneously with the first one. The driver must function correctly even when being executed concurrently by two or more CPUs, which implies that all sensitive data structures may only be accessed from inside critical regions. Just assuming that there will not be any other calls until the current one is finished is not permitted.

Finally, conformant drivers must work not only on Windows 2000 but also on Windows 98. It may be necessary to recompile the driver on each system however, and use of C preprocessor commands to isolate platform dependencies is permitted.

In UNIX, drivers are located by using their major device numbers. Windows 2000 uses a different scheme. At boot time, or when a new hot pluggable plug-and-play device is attached to the computer, Windows 2000 automatically detects it and calls the plug-and-play manager. The manager queries the device to find out what the manufacturer and model number are. Equipped with this knowledge, it looks on the hard disk in a certain directory to see if it has the driver. If it does not, it displays a dialog box asking the user to insert a floppy disk or CD-ROM with the driver. Once the driver is located, it is loaded into memory.

Each driver must supply a set of procedures that can be called to get its services. The first one, called *DriverEntry*, initializes the driver. It is called just after the driver is loaded. It may create tables and data structures, but must not touch the device yet. It also fills in some of the fields of the driver object created by the I/O manager when the driver was loaded. The fields in the driver object include pointers to all the other procedures that drivers must supply. In addition, for each device controlled by the driver (e.g., each IDE disk controlled by the IDE disk driver), a **device object** is created and initialized to point to the driver object. These driver objects are entered into a special directory, *\??*. Given a device object, the driver object can be located easily, and hence its methods can be called.

A second required procedure is *AddDevice*, which is called once (by the plug-and-play manager) for each device to be added. Once this has been accomplished, the driver is called with the first IRP, which sets up the interrupt vector and actually initializes the hardware. Other procedures that drivers must contain

are the interrupt service procedure, various timer management procedures, a fast I/O path, DMA control, a way to cancel currently executing requests, and many more. All in all, Windows 2000 drivers are so complex that multiple books have been written about them (Cant, 1999; Oney, 1999; and Viscarola and Mason, 1999).

A driver in Windows 2000 may do all the work by itself, as the printer driver does in Fig. 11-30 (just as an example). 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. 11-30.

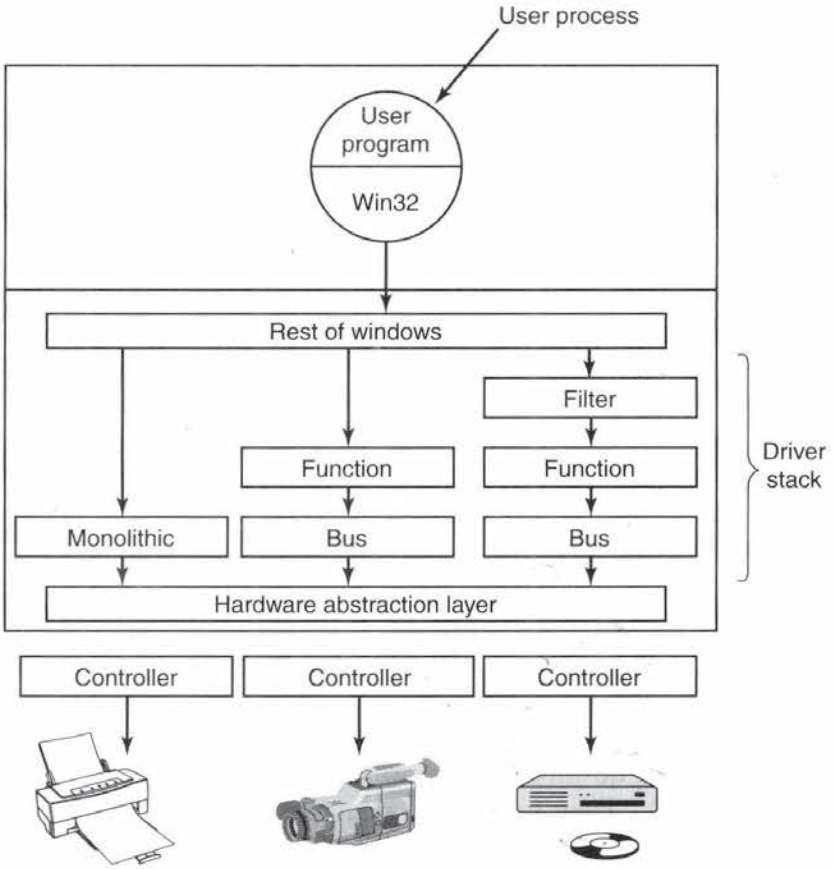


Figure 11-30. Windows 2000 allows drivers to be stacked.

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

Another use of stacking drivers is to be able to insert **filter drivers** into the stack. A filter driver performs some transformation on the data on the way up or down. 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 have to be aware of it and it works automatically for all data going to (or coming from) the device.

11.7 THE WINDOWS 2000 FILE SYSTEM

Windows 2000 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. FAT-32 uses 32-bit disk addresses and supports disk partitions up to 2 TB. NTFS is a new file system developed specifically for Windows NT and carried over to Windows 2000. It uses 64-bit disk addresses and can (theoretically) support disk partitions up to 2^{64} bytes, although other considerations limit it to smaller sizes. Windows 2000 also supports read-only file systems for CD-ROMs and DVDs. It is possible (even common) to have the same running system have access to multiple file system types available at the same time.

In this chapter we will treat the NTFS file system because it is a modern file system unencumbered by the need to be fully compatible with the MS-DOS file system, which was based on the CP/M file system designed for 8-inch floppy disks more than 20 years ago. Times have changed and 8-inch floppy disks are not quite state of the art any more. Neither are their file systems. Also, NTFS differs both in user interface and implementation in a number of ways from the UNIX file system, which makes it a good second example to study. NTFS is a large and complex system and space limitations prevent us from covering all of its features, but the material presented below should give a reasonable impression of it.

11.7.1 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*). Unfortunately, the Win32 API does not fully support case-sensitivity for file names and not at all for directory names, so this advantage is lost to programs restricted to using Win32 (e.g., for Windows 98 compatibility).

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 of which is 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 was borrowed from the Apple Macintosh, in which files have two streams, the data fork and the resource fork. This concept was incorporated into NTFS to allow an NTFS server be able to serve Macintosh clients.

File streams can be used for purposes other than Macintosh compatibility. For example, a photo editing program could use the unnamed stream for the main image and a named stream for a small thumbnail version. This scheme is simpler than the traditional way of putting them in the same file one after another. Another use of streams is in word processing. These programs often make two versions of a document, a temporary one for use during editing and a final one when the user is done. By making the temporary one a named stream and the final one the unnamed stream, both versions automatically share a file name, security information, timestamps, etc. with no extra work.

The maximum stream length is 2^{64} bytes. To get some idea of how big a 2^{64}-byte stream is, imagine that the stream were written out in binary, with each of the 0s and 1s in each byte occupying 1 mm of space. The 2^{67}-mm listing would be 15 light-years long, reaching far beyond the solar system, to Alpha Centauri and back. File pointers are used to keep track of where a process is in each stream, and these are 64 bits wide to handle the maximum length stream, which is about 18.4 exabytes.

The Win32 API function calls for file and directory manipulation are roughly similar to their UNIX counterparts, except most have more parameters and the security model is different. Opening a file returns a handle, which is then used for reading and writing the file. For graphical applications, no file handles are predefined. Standard input, standard output, and standard error have to be acquired explicitly if needed; in console mode they are preopened, however. Win32 also has a number of additional calls not present in UNIX.

11.7.2 File System API Calls in Windows 2000

The principal Win32 API functions for file management are listed in Fig. 11-31. There are actually many more, but these give a reasonable first impression of the basic ones. Let us now examine these calls briefly. CreateFile can be used to create a new file and return a handle to it. This API function must also be used to open existing files as there is no FileOpen API function. We have not listed the

Win32 API function	UNIX	Description
CreateFile	open	Create a file or open an existing file; return a handle
DeleteFile	unlink	Destroy an existing file
CloseHandle	close	Close a file
ReadFile	read	Read data from a file
WriteFile	write	Write data to a file
SetFilePointer	lseek	Set the file pointer to a specific place in the file
GetFileAttributes	stat	Return the file properties
LockFile	fcntl	Lock a region of the file to provide mutual exclusion
UnlockFile	fcntl	Unlock a previously locked region of the file

Figure 11-31. The principal Win32 API functions for file I/O. The second column gives the nearest UNIX equivalent.

parameters for the API functions because they are so voluminous. As an example, CreateFile has seven parameters, which are roughly summarized as follows:

1. A pointer to the name of the file to create or open.
2. Flags telling whether the file can be read, written, or both.
3. Flags telling whether multiple processes can open the file at once.
4. A pointer to the security descriptor, telling who can access the file.
5. Flags telling what to do if the file exists/does not exist.
6. Flags dealing with attributes such as archiving, compression, etc.
7. The handle of a file whose attributes should be cloned for the new file.

The next six API functions in Fig. 11-31 are fairly similar to the corresponding UNIX system calls. The last two allow a region of a file to be locked and unlocked to permit a process to get guaranteed mutual exclusion to it.

Using these API functions, it is possible to write a procedure to copy a file, analogous to the UNIX version of Fig. 6-5. Such a code fragment (without any error checking) is shown in Fig. 11-32. It has been designed to mimic our UNIX version. In practice, one would not have to program a copy file program since CopyFile is an API function (which executes something close to this program as a library procedure).

Windows 2000 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 MS-DOS. There is a concept of a current working directory and path names can be relative or absolute. Hard and symbolic links are supported,

```
/* Open files for input and output. */
inhandle = CreateFile("data", GENERIC_READ, 0, NULL, OPEN_EXISTING, 0, NULL);
outhandle = CreateFile("newf", GENERIC_WRITE, 0, NULL, CREATE_ALWAYS,
    FILE_ATTRIBUTE_NORMAL, NULL);

/* Copy the file. */
do {
    s = ReadFile(inhandle, buffer, BUF_SIZE, &count, NULL);
    if (s && count > 0) WriteFile(outhandle, buffer, count, &ocnt, NULL);
} while (s > 0 && count > 0);

/* Close the files. */
CloseHandle(inhandle);
CloseHandle(outhandle);
```

Figure 11-32. A program fragment for copying a file using the Windows 2000 API functions.

the former implemented by having multiple directory entries, as in UNIX, and the latter implemented using reparse points (discussed later in this chapter). In addition, compression, encryption, and fault tolerance are also supported. These features and their implementations will be discussed later in this chapter.

The major directory management API functions are given in Fig. 11-33, again along with their nearest UNIX equivalents. The functions should be self explanatory.

Win32 API function	UNIX	Description
CreateDirectory	mkdir	Create a new directory
RemoveDirectory	rmdir	Remove an empty directory
FindFirstFile	opendir	Initialize to start reading the entries in a directory
FindNextFile	readdir	Read the next directory entry
MoveFile	rename	Move a file from one directory to another
SetCurrentDirectory	chdir	Change the current working directory

Figure 11-33. The principal Win32 API functions for directory management. The second column gives the nearest UNIX equivalent, when one exists.

11.7.3 Implementation of the Windows 2000 File System

NTFS is a highly complex and sophisticated file system. It was designed from scratch, rather than being an attempt to improve the old MS-DOS file system. Below we will examine a number of its features, starting with its structure, then moving on to file name lookup, file compression, 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 main 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 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 other 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. 11-34. 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 because 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.

The first 16 MFT records are reserved for NTFS metadata files, as shown in Fig. 11-34. Each of the records 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 the system can find the MFT file. Clearly, Windows 2000 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 at system installation time.

Record 1 is a duplicate of the early part 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 goes bad. 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. Changes to file attributes are also logged here. In fact, the only changes not

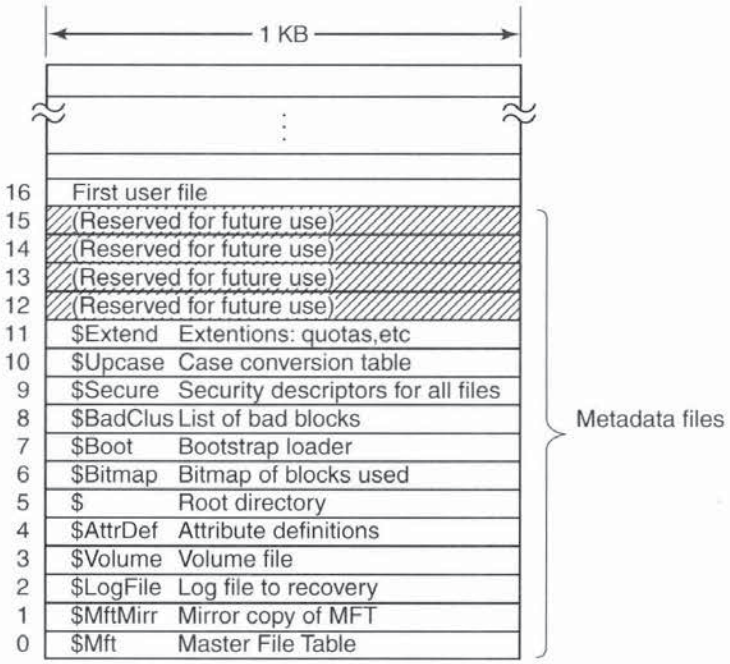


Figure 11-34. The NTFS master file table.

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 4 MFT records are reserved for future use.

Each MFT record consists of a record header followed by a sequence of (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. Following the record header comes the header of the first attribute, then the first attribute value, the second attribute header, the second attribute value, and so on.

NTFS defines 13 attributes that can appear in MFT records. These are listed in Fig. 11-35. Each MFT record consists of a sequence of attribute headers, each of which identifies the attribute it is heading 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 a separate disk block. 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.

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 11-35. The attributes used in MFT records.

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, etc. It is a fixed-length field and is always present. The file name is variable length in Unicode. In order to make files with nonMS-DOS names accessible to old 16-bit programs, files can also have an 8 + 3 MS-DOS name. If the actual file name conforms to the MS-DOS 8 + 3 naming rule, a secondary MS-DOS name is not used.

In NT 4.0, security information could be put in an attribute, but in Windows 2000 it all goes into a single file so that multiple files can share the same security

descriptions. 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.

The object ID attribute gives the file a unique name. This is sometimes needed internally. The reparse point tells the procedure parsing the file name to do something special. This mechanism is used for mounting and symbolic links. The two volume attributes are only used 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 everyone has been waiting for: the data. The stream name, if present, goes in this attribute header. Following the header is either a list of disk addresses telling which blocks the file contains, or for files of only a few hundred bytes (and there are many of these), the file itself. Putting the actual file data in the MFT record is called an **immediate file** (Mullender and Tanenbaum, 1987).

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.

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 file 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, if possible.

The blocks in a file are described by a sequence of records, each one describing a sequence of logically contiguous blocks. For a file with no holes in it, there will be only one such record. Files that are written in order from beginning to end all belong in this category. For a file with one hole in it (e.g., only blocks 0–49 and blocks 60–79 are defined), there will be two records. Such a file 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.

Each record begins with a header giving the offset of the first block within the file. 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 file is illustrated in Fig. 11-36.

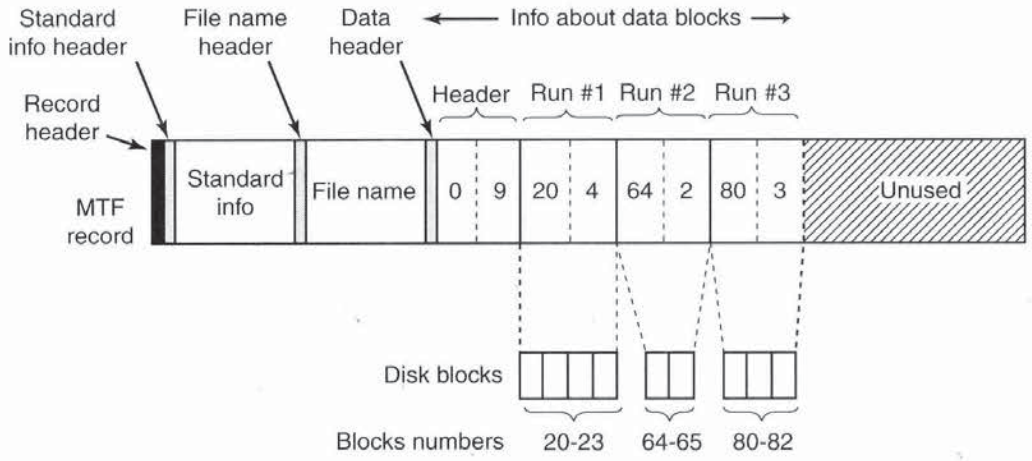


Figure 11-36. An MFT record for a three-run, nine-block file.

In this figure we have an MFT record for a short file (short here means that all the information about the file blocks fits in one MFT record). 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 good a job the disk block allocator did in finding runs of consecutive blocks when the file was created. For a n-block file, the number of runs can be anything from 1 up to and including n.

Several comments are worth making here. First, there is no upper limit to the size of files 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 file consisting of 20 separate runs of 1 million 1-KB blocks each fits easily in one MFT record, whereas a 60-KB file 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. 11-37 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.

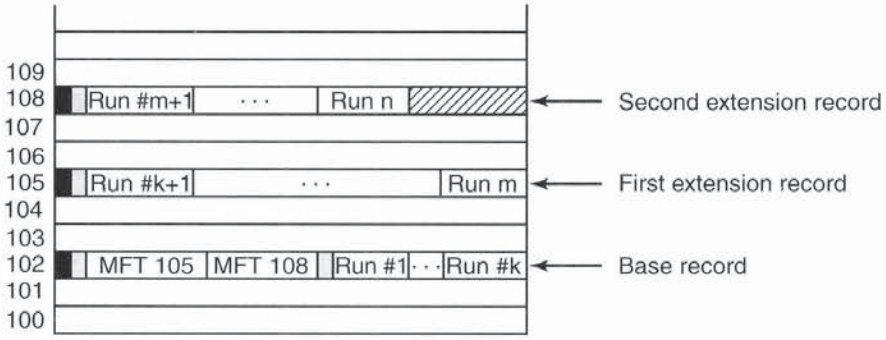


Figure 11-37. A file that requires three MFT records to store all its runs.

Note that Fig. 11-37 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 only necessary to examine 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 on disk 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. 11-38. 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.

File Name Lookup

We now have enough information to see how file name lookup occurs. When a user program wants to open a file, it typically makes a call like

```
CreateFile("C:\maria\web.htm", ...)
```

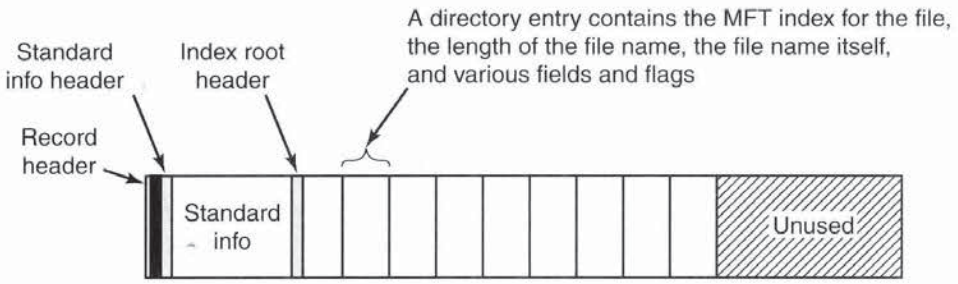


Figure 11-38. The MFT record for a small directory.

This call goes to the user-level shared library, *kernel32.dll*, where *\??* is prepended to the file name giving

```
\??\C:\maria\web.htm
```

It is this name that is passed as a parameter to the system call NtFileCreate.

Then the operating system starts the search at the root of the object manager's name space (see Fig. 11-12). It then looks in the directory *\??* to find *C:*, which it will find. This file is a symbolic link to another part of the object manager's name space, the directory *\Device*. The link typically ends at an object whose name is something like *\Device\HarddiskVolume1*. This object corresponds to the first partition of the first hard disk. From this object it is possible to determine which MFT to use, namely the one on this partition. These steps are shown in Fig. 11-39.

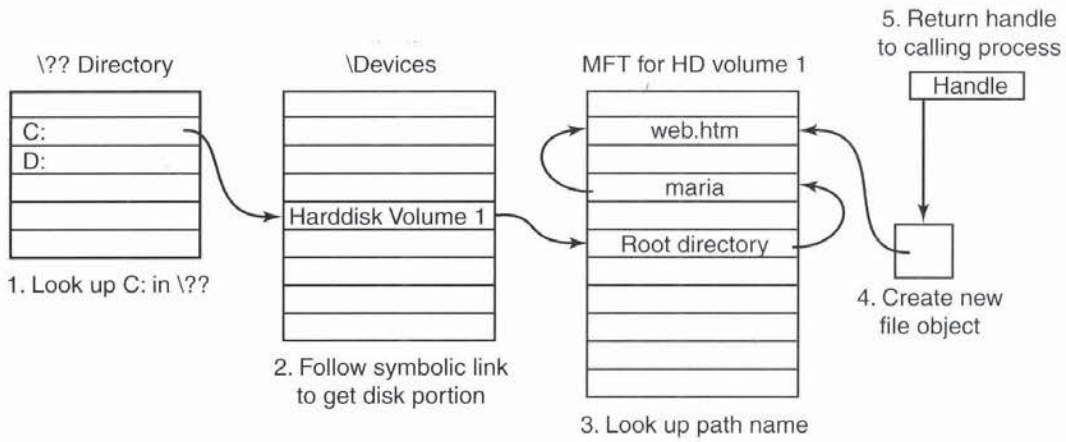


Figure 11-39. Steps in looking up the file *C:\maria\web.htm*.

The parsing of the file name continues now at the root directory, whose blocks can be found from entry 5 in the MFT (see Fig. 11-34). The string "maria" is now looked up in the root directory, which returns the index into the MFT for the directory *maria*. This directory is then searched for the string "web.htm". If

successful, the result is a new object created by the object manager. The object, which is unnamed, contains the index of the MFT record for the file. A handle to this object is returned to the calling process. On subsequent ReadFile calls, the handle is provided, which allows the object manager to find the index and then the contents of the MFT record for the file. If a thread in a second process opens the file again, it gets a handle to a new file object.

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**. It is possible to tag a file or directory as a reparse point and associate a block of data with it. When the file or directory is encountered during a file name parse, exception processing is triggered and the block of data is interpreted. It can do various things, including redirecting the search to a different part of the directory hierarchy or even to a different partition. This mechanism is used to support both symbolic links and mounted file systems.

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 of the fact 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 data can be stored in 15 or fewer blocks, the compressed data 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 less, and so on.

Figure 11-40(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. 11-40(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 are not. It sees that 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 it contains the boot sector, using it for data is impossible anyway.

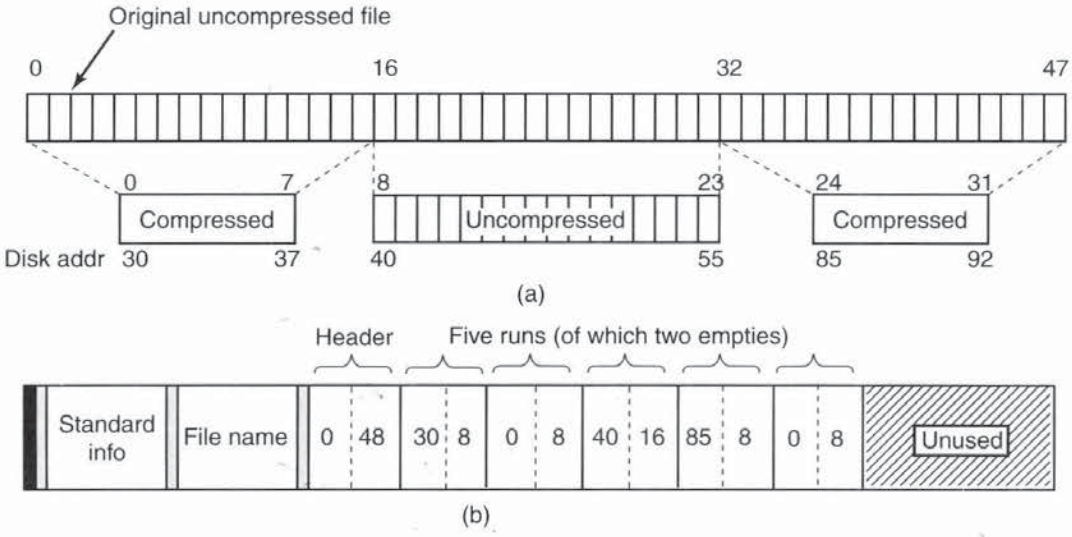


Figure 11-40. (a) An example of a 48-block file being compressed to 32 blocks. (b) The MFT record for the file after compression.

Random access to compressed files is possible, but tricky. Suppose that a process does a seek to block 35 in Fig. 11-40. 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.

File Encryption

Computers are used nowadays to store all kinds of sensitive data, including plans for corporate takeovers, tax information, and love letters (love email?), which the owners do not especially want revealed to anyone. Information loss can happen when a laptop computer is lost or stolen, a desktop system is rebooted using an MS-DOS floppy disk to bypass Windows 2000 security, or a hard disk is physically removed from one computer and installed on another one with an insecure operating system. Even the simple act of going to the bathroom and leaving the computer unattended and logged in can be a huge security breach.

Windows 2000 addresses these problem by having an option to encrypt files, so even in the event the computer is stolen or rebooted using MS-DOS, the files will be unreadable. The normal way to use Windows 2000 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 is not done by NTFS itself, but by a driver called **EFS** (**Encrypting File System**), which is positioned between NTFS and the user

process. In this way, application programs are unaware of encryption and NTFS itself is only partially involved in it.

To understand how the encrypting file system works, it is necessary to understand how modern cryptography works. For this purpose, a brief review was given in Sec. 9.2. Readers not familiar with the basics of cryptography should read that section before continuing.

Now let us see how Windows 2000 encrypts files. When the user asks a file to be encrypted, a random 128-bit file key is generated and used to encrypt the file block by block using a symmetric algorithm parametrized by this key. Each new file encrypted gets a different 128-bit random file key, so no two files use the same encryption key, which increases security in case one key is compromised. The current encryption algorithm is a variant of **DES (Data Encryption Standard)**, but the EFS architecture supports the addition of new algorithms in the future. Encrypting each block independently of all the others is necessary to make random access still possible.

The file key has to be stored somewhere so the file can be decrypted later. If it were just stored on the disk in plaintext, then someone who stole or found the computer could easily decrypt the file, defeating the purpose of encrypting the files. For this reason, the file keys must all be encrypted before they are stored on the disk. Public-key cryptography is used for this purpose.

After the file is encrypted, the location of the user's public key is looked up using information in the registry. There is no danger of storing the public key's location in the registry because if a thief steals the computer and finds the public key, there is no way to deduce the private key from it. The 128-bit random file key is now encrypted with the public key and the result stored on disk along with the file, as shown in Fig. 11-41.

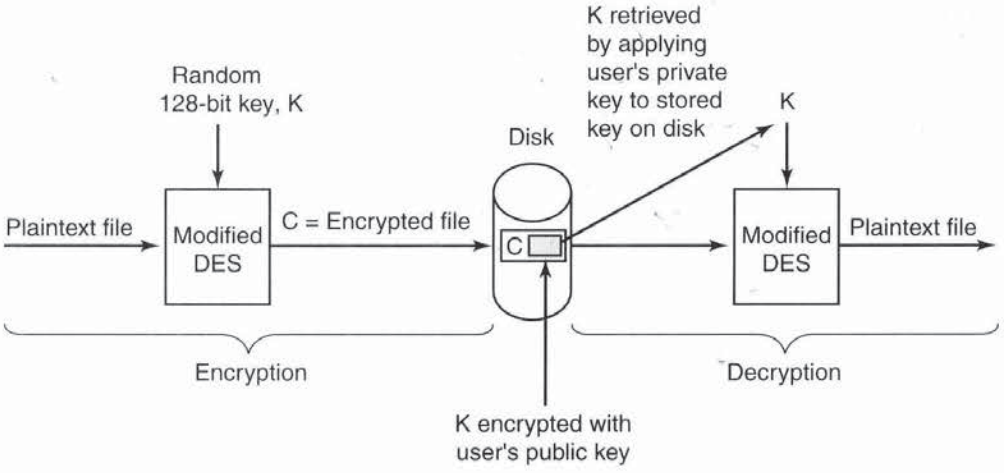


Figure 11-41. Operating of the encrypting file system.

To decrypt a file, the encrypted 128-bit random file key is fetched from disk. However, to decrypt it and retrieve the file key, the user must present the private

key. Ideally, this should be stored on a smart card, external to the computer, and only inserted in a reader when a file has to be decrypted. Although Windows 2000 supports smart cards, it does not store private keys on them.

Instead, the first time a user encrypts a file using EFS, Windows 2000 generates a (private key, public key) pair and stores the private key on disk encrypted using a symmetric encryption algorithm. The key used for the symmetric algorithm is derived either from the user's login password or from a key stored on the smart card, if smart card login is enabled. In this way, EFS can decrypt the private key at login time and keep it within its own virtual address space during normal operation so it can decrypt the 128-bit file keys as needed without further disk accesses. When the computer is shut down, the private key is erased from EFS' virtual address space so anyone stealing the computer will not have access to the private key.

A complication occurs when multiple users need access to the same encrypted file. Currently the shared use of encrypted files by multiple users is not supported. However, the EFS architecture could support sharing in the future by encrypting each file's key multiple times, once with the public key of each authorized user. All of these encrypted versions of the file key could be attached to the file.

The potential need to share encrypted files is one reason why this two-key system is used. If all files were encrypted by their owner's key, there would be no way to share any files. By using a different key to encrypt each file, this problem can be solved.

Having a random file key per file but encrypting it with the owner's symmetric key does not work because having the symmetric encryption key just lying around in plain view would ruin the security— generating the decryption key from the encryption key is too easy. Thus (slow) public-key cryptography is needed to encrypt the file keys. Because the encryption key is public anyway, having it lying around is not dangerous.

The other reason the two-key system is used is performance. Using public-key cryptography to encrypt each file would be too slow. It is much more efficient to use symmetric-key cryptography to encrypt the data and public-key cryptography to encrypt the symmetric file key.

11.8 SECURITY IN WINDOWS 2000

Having just looked at encryption in the file system, this is a good time to examine security in general. NT was designed to meet the U.S. Department of Defense's C2 security requirements (DoD 5200.28-STD), the Orange Book, which we studied in Chap. 9. 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 2000 was not specifically designed for C2 compliance, it inherits many security properties from 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 (none of which are met by Windows 98, incidentally).

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 2000 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.

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 a stack grows, the pages mapped in are initialized to zero so processes cannot find any old information put there by the previous owner (hence the zeroed page list in Fig. 11-27, 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.

In the next section we will describe the basic concepts behind Windows 2000 security. After that we will look at the security system calls. Finally, we will conclude by seeing how security is implemented.

11.8.1 Fundamental Concepts

Every Windows 2000 user (and group) is identified by a **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 its SID and other properties. It is normally assigned at login time by *winlogon* and is shown in Fig. 11-42, although processes should call *GetTokenInformation* to acquire this information since it may change in the future. 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* fields specify the groups to which the process belongs; this is needed for POSIX conformance. 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, 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.

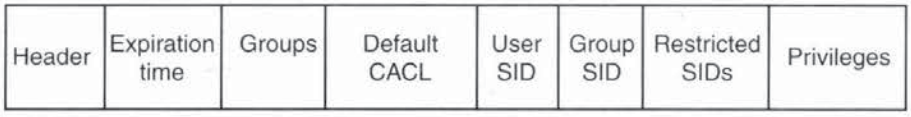


Figure 11-42. Structure of an access token .

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 case the thread's access token overrides the process' access token. In particular, a client thread can pass its access token to a server thread to allow the server to access the client's protected files and other objects. This mechanism is called **impersonation**.

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. A security descriptor consists of a header followed by a DACL with one or more **ACE**s (**Access Control Elements**). The two main kinds of elements are Allow and Deny. An allow element specifies a SID and a bitmap that specifies which operations processes with 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. 11-43. The SID Everyone refers to the set of all users, but it is overridden by any explicit ACEs that follow.

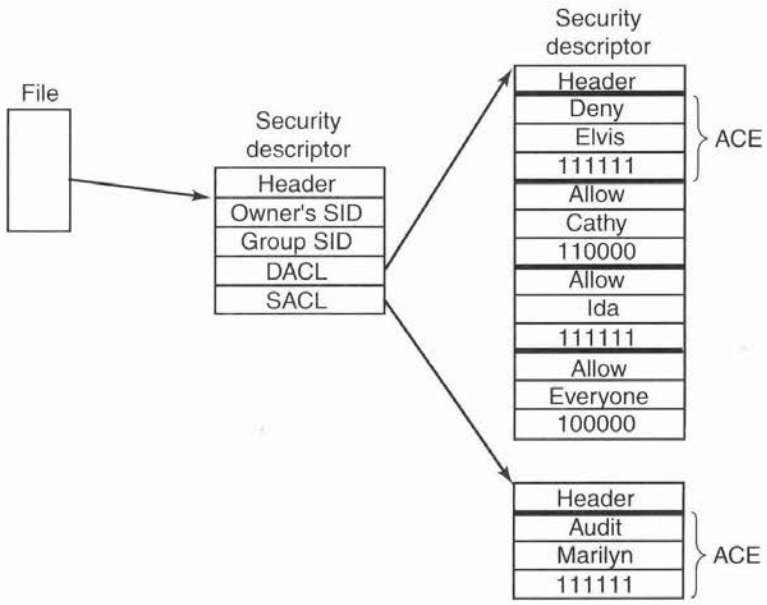


Figure 11-43. An example security descriptor for a file.

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 system-wide security event log. In Fig. 11-43, every operation that Marilyn performs on the file will be logged. Windows 2000 provides additional auditing features to log sensitive accesses.

11.8.2 Security API Calls

Most of the Windows 2000 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. 11-43. If no security descriptor is provided in the object creation call, the default security in the caller's access token (see Fig. 11-42) 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. 11-44. To create a security descriptor, storage for it is first allocated and then initialized using InitializeSecurityDescriptor. This call fills in the header. If the

owner SID is not known, it can be looked up by name using LookupAccountSid. It can then be inserted into the security descriptor. The same holds for the group SID, if any. Normally, these will be the caller's own SID and one of the caller's groups, but the system administrator can fill in any SIDs.

Win32 API function	Description
InitializeSecurityDescriptor	Prepare a new security descriptor for use
LookupAccountSid	Look up the SID for a given user name
SetSecurityDescriptorOwner	Enter the owner SID in the security descriptor
SetSecurityDescriptorGroup	Enter a group SID in the security descriptor
InitializeAcl	Initialize a DACL or SACL
AddAccessAllowedAce	Add a new ACE to a DACL or SACL allowing access
AddAccessDeniedAce	Add a new ACE to a DACL or SACL denying access
DeleteAce	Remove an ACE from a DACL or SACL
SetSecurityDescriptorDacl	Attach a DACL to a security descriptor

Figure 11-44. The principal Win32 API functions for security.

At this point the security descriptor's DACL (or SACL) can be initialized with InitializeAcl. ACL entries can be added using AddAccessAllowedAce, and AddAccessDeniedAce. These calls can be repeated multiple times to add as many ACE entries as are needed. DeleteAce can be used to remove an entry, more like on an existing ACL than on one being constructed for the first time. When the ACL is ready, SetSecurityDescriptorDacl can be used to attach it to the security descriptor. Finally, when the object is created, the newly minted security descriptor can be passed as a parameter to have it attached to the object.

11.8.3 Implementation of Security

Security in a standalone Windows 2000 system is implemented by a number of components, most of which we have already seen (networking is a whole other story and beyond the scope of this book). Logging in is handled by *winlogon* and authentication is handled by *lsass* and *msgina.dll* as discussed in Sec. 11.4.5. The result of a successful login is a new shell with its associated access token. This process uses the SECURITY and SAM keys in the registry. The former sets the general security policy and the latter contains the security information for the individual users, as discussed in Sec. 11.2.3.

Once a user is logged in, security operations happen when an object is opened for access. Every OpenXXX call requires the name of the object being opened and the set of rights needed. During processing of the open, the security manager (see Fig. 11-7) checks to see if the caller has all the rights required. It performs this check by looking at the caller's access token and the DACL associated with the

object. It goes down the list of entries in the ACL in order. As soon as it finds an entry that matches the caller's SID or one of the caller's groups, the access found there is taken as definitive. If all the rights the caller needs are available, the open succeeds; otherwise it fails.

DACLs can have Deny entries as well as Allow entries, as we have seen. For this reason, it is usual to put entries denying access ahead of entries granting access in the ACL, so that a user who is specifically denied access cannot get in via a back door by being a member of a group that has legitimate access.

After an object has been opened, a handle to it is returned to the caller. On subsequent calls, the only check that is made is whether the operation now being tried was in the set of operations requested at open time, to prevent a caller from opening a file for reading and then trying to write on it. Any log entries required by the SACL are made.

11.9 CACHING IN WINDOWS 2000

The Windows 2000 cache manager does caching for performance reasons, conceptually similar to caches in other operating systems. However, its design has some unusual properties that are worth looking at briefly.

The cache manager's job is to keep file system blocks that have been used recently in memory to reduce access time on any subsequent reference. Windows 2000 has a single integrated cache that works for all the file systems in use, including NTFS, FAT-32, FAT-16, and even CD-ROM file systems. This means that the file systems do not need to maintain their own caches.

As a consequence of the design goal to have a single integrated cache despite the presence of multiple file systems, the cache manager is located in an unusual position in the system, as we saw in Fig. 11-7. It is not part of the file system because there are multiple independent file systems that may have nothing in common. Instead it operates at a higher level than the file systems, which are technically drivers under control of the I/O manager.

The Windows 2000 cache is organized by virtual block, not physical block. To see what this means, remember that traditional file caches keep track of blocks by two-part addresses of the form (partition, block), where the first member denotes a device and partition and the second member is a block number within that partition. The Windows 2000 cache manager does not do that. Instead, it uses (file, offset) to refer to a block. (Technically, streams are cached rather than files, but we will ignore that detail below.)

The reason for this unorthodox arrangement is that when a request comes in to the cache manager, it is specified as (file, offset) because that is all the calling process knows. If cache blocks had been labeled by a (partition, block) tag, the cache manager would have no way of knowing which (file, offset) block cor-

responds to which (partition, block) block since it is the file systems that maintain those mappings.

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. The total amount of virtual address space the cache manager can use is determined at boot time and depends on the amount of RAM present. 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 copied is not in physical memory, a page fault will occur and the memory manager will satisfy the fault in the usual away. The cache manager is not even aware of whether the block was in the cache or not. The copy always succeeds.

The operation of the cache manager is shown in Fig. 11-45 in the case of an NTFS file system on a SCSI disk and a FAT-32 file system on an IDE disk. When a process does a read on a file, the request is directed to the cache manager. If the block needed is in the cache, it is copied to the user immediately. If it is not in the cache, the cache manager gets a page fault when trying to copy it. When the page fault has been handled, the block is copied to the calling process.

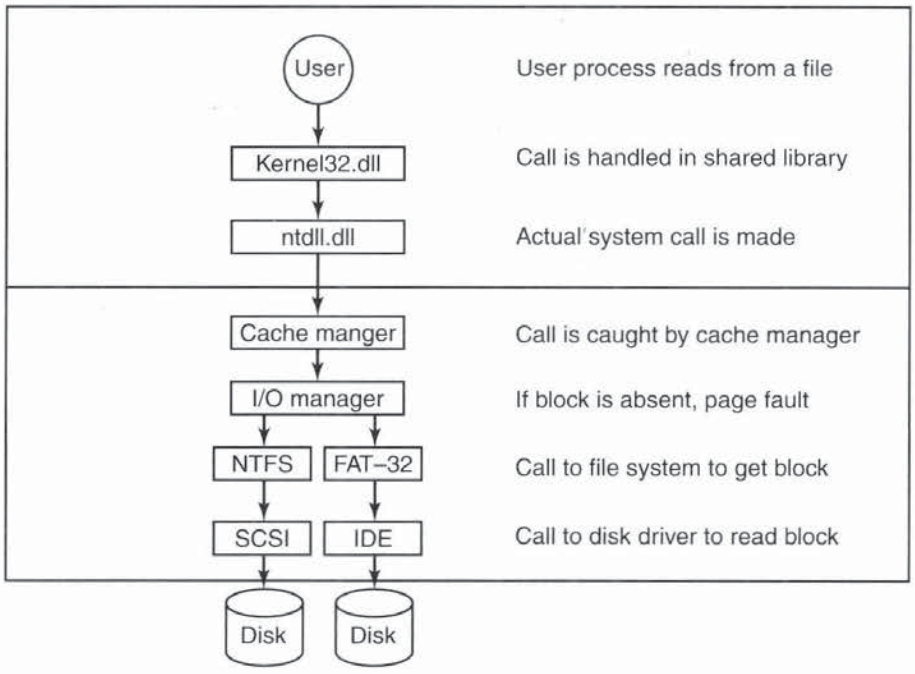


Figure 11-45. The path through the cache to the hardware.

As a consequence of this design, the cache manager does not know how many of its mapped pages are in physical memory or even how large its cache is. Only

the memory manager knows for sure. This approach allows the memory manager to dynamically trade off the size of the cache against memory for user pages. If there is little file activity but there are many processes active, the memory manager can use most of physical memory for process pages. On the other hand, if there is a lot of file activity and few processes, more physical memory can be devoted to the cache.

Another property the cache manager has is that coherence is maintained between memory-mapped files and files that are open for reading and writing. Consider, for example, a situation in which one process opens some file for reading and writing and a second process maps that file onto its address space. What happens if the second process writes on the file directly and immediately thereafter the first one reads in a block that was just changed? Does it get stale data?

The answer is no. In both cases—open files and mapped files— the cache manager maps a 256-KB piece of its virtual address space onto the file. The file is only mapped once, no matter how many processes have it open or mapped. In the case of a mapped file, both the cache manager and the user process share pages in memory. When a read request has to be satisfied, the cache manager just copies the page from memory to the user buffer, which always reflects the exact current state of the file because the cache manager is using the same pages as the process that has the file mapped in.

11.10 SUMMARY

Windows 2000 is structured in the HAL, the kernel, the executive, and a thin system services layer that catches incoming system calls. In addition, there are a variety of device drivers, including the file systems and GDI. The HAL hides certain differences in hardware from the upper layers. The kernel attempts to hide the remaining differences from the executive to make it almost completely machine independent.

The executive is based on memory objects. User processes create them and get back handles to manipulate them later. Executive components can also create objects. The object manager maintains a name space into which objects can be inserted for subsequent lookup.

Windows 2000 supports processes, jobs, threads, and fibers. Processes have virtual address spaces and are containers for resources. Threads are the unit of execution and are scheduled by the operating system. Fibers are lightweight threads that are scheduled entirely in user space. Jobs are collections of processes and are used for assigning resource quotas. Scheduling is done using a priority algorithm in which the highest priority ready thread runs next.

Windows 2000 supports a demand-paged virtual memory. The paging algorithm is based on the working set concept. The system maintains several lists of

free pages, so that when a page fault occurs, there is generally a free page available. The free page lists are fed by trimming the working sets using complex formulas that try to throw out pages that have not been used in a long time.

I/O is done by device drivers, which follow the Windows Device Model. Each driver starts out by initializing a driver object that contains the addresses of the procedures that the system can call to add devices or perform I/O. Drivers can be stacked to act as filters.

The NTFS file system is based on a master file table, which has one record per file or directory. Each file has multiple attributes, which can either be in the MFT record or nonresident, on the disk. NTFS supports compression and encryption, among other features.

Security is based on access control lists. Each process has an access control token that tells who it is and what special privileges it has, if any. Each object has a security descriptor associated with it. The security descriptor points to a discretionary access control list that contains access control entries that can allow or deny access to individuals or groups.

Finally, Windows 2000 maintains a single system-wide cache for all the file systems. It is a virtual cache rather than a physical one. Requests for disk blocks go first to the cache. If they cannot be satisfied, then the appropriate file system is called to fetch the needed blocks.

PROBLEMS

1. Give one advantage and one disadvantage of the registry versus having individual *.ini* files?
2. A mouse can have 1, 2, or 3 buttons. All three types are in use. Does the HAL hide this difference from the rest of the operating system? Why or why not?
3. The HAL keeps track of time starting in the year 1601. Give an example of an application where this feature is useful.
4. The POSIX subsystem needs to implement UNIX-style signals. If a user hits the key for the Quit signal, is this scheduled as a DPC or an APC?
5. Many components of the executive (Fig. 11-7) call other components of the executive. Give three examples of one component calling another one, but use (six) different components in all.
6. Win32 does not have signals. If they were to be introduced, they could be per process, per thread, both, or neither. Make a proposal and explain why it is a good idea.
7. An alternative to using DLLs is to statically link each program with precisely those library procedures it actually calls, no more and no less. If this scheme were to be introduced, would it make more sense on client machines or on server machines?

8. The file *ntdll.dll* exports 1179 function calls while *ntoskrnl.exe* exports 1209 function calls. Is this a bug? What might be the cause of this discrepancy?

9. Objects managed by the object manager are variable-sized, with different objects having different sizes. Can an object start at an arbitrary byte in the nonpaged pool? *Hint*: You do not need any information about Windows 2000 other than what was given in the text.

10. Is there any limit on the number of different operations that can be defined on an executive object? If so, where does this limit come from? If not, why not?

11. The Win32 API call WaitForMultipleObjects allows a thread to block on a set of synchronization objects whose handles are passed as parameters. As soon as any one of them is signaled, the calling thread is released. Is it possible to have the set of synchronization objects include two semaphores, one mutex, and one critical section? Why or why not? *Hint*: This is not a trick question but it does require some careful thought.

12. Name three reasons why a process might be terminated.

13. Consider the situation of Fig. 11-19 in which the system is about to schedule a thread. Assuming that each thread is compute bound, how long does it take before some thread at priority 3 gets to run on Windows 2000 Professional?

14. Suppose that the quantum is set to 20 msec and the current thread, at priority 24, has just started a quantum. Suddenly an I/O operation completes and a priority 28 thread is made ready. About how long does it have to wait to get serviced?

15. In Windows 2000, the current priority is always greater than or equal to the base priority. Are there any circumstances in which it would make sense to have the current priority be lower than the base priority? If so, give an example. If not, why not?

16. Some MS-DOS programs were written in assembly language using instructions such as IN port and OUT port. Can such programs be run under Windows 2000? If not, can you think of a way they could be supported?

17. Name two ways to give better response time to important processes.

18. Position-independent code was briefly discussed in the text. It is a technique for allowing two processes to share the same procedure at different virtual addresses. Can this problem be solved by just setting the page tables of the two processes appropriately? Explain your answer.

19. Shared libraries in Windows 2000 are contained in *.dll* files that multiple processes may map in at the same time. If two processes need to map in the same shared library at different virtual addresses, a problem will occur. How could this problem be solved on the Pentium using a property of its memory architecture? If the solution requires changes to Windows 2000 to implement, state what changes are needed.

20. If a region of virtual address space is reserved but not committed, do you think a VAD is created for it? Defend your answer.

21. Which of the transitions shown in Fig. 11-27 are policy decisions, as opposed to required moves forced by system events (e.g., a process exiting and freeing its pages)?

22. Suppose that a page is shared and in two working sets at once. If it is evicted from one of the working sets, where does it go in Fig. 11-27?

23. When a process unmaps a clean page, it makes the transition (5) in Fig. 11-27. Why is there no transition to the modified list when a dirty page is unmapped?

24. Assuming that both RAM and video RAM each can read or write a 32-bit word in 10 nsec, how long does it take to paint the background on an XGA screen in the best case?

25. A file has the following mapping. Give the MFT run entries.

Offset	0	1	2	3	4	5	6	7	8	9	10
Disk address	50	51	52	22	24	25	26	53	54	-	60

26. Consider the MFT record of Fig. 11-36. Suppose that the file grew and a 10th block was assigned to the end of the file. The number of this block is 66. What would the MFT record look like now?

27. In Fig. 11-40(b), the first two runs are each of length 8 blocks. Is it just an accident they are equal or does this have to do with the way compression works? Explain your answer.

28. Suppose that you wanted to build Windows 2000 Lite. Which of the fields of Fig. 11-42 could be removed without weakening the security of the system?

29. The *regedit* command can be used to export part or all of the registry to a text file under all current versions of Windows. Save the registry several times during a work session and see what changes. If you have access to a Windows computer on which you can install software or hardware, find out what changes when a program or device are added or removed.

30. Write a UNIX program that simulates writing an NTFS file with multiple streams. It should accept a list of one or more files as arguments and write an output file that contains one stream with the attributes of all arguments and additional streams with the contents of each of the arguments. Now write a second program for reporting on the attributes and streams and extracting all the components.

31. Write a program that generates an MS-DOS 8 + 3 filename given any file name. Use the Windows 2000 algorithm.

12

OPERATING SYSTEM DESIGN

In the past 11 chapters, we have covered a lot of ground and taken a look at many concepts and examples relating to operating systems. But studying existing operating systems is different from designing a new one. In this chapter we are going to take a quick look at some of the issues and trade-offs that operating systems designers have to take into account when designing and implementing a new system.

There is a certain amount of folklore about what is good and what is bad floating around in the operating systems community, but surprisingly little has been written down. Probably the most important book is Fred Brooks' classic (1975) book *The Mythical Man Month* in which he relates his experiences in designing and implementing IBM's OS/360. The 20th anniversary edition revises some of that material and adds four new chapters (Brooks, 1995). Probably the only operating systems textbook dealing with design in a serious way is *Operating Systems: A Design-Oriented Approach* (Crowley, 1997).

Three classic papers on operating system design are "Hints for Computer System Design" (Lampson, 1984), "On Building Systems that Will Fail" (Corbató, 1991), and "End-to-End Arguments in System Design" (Saltzer et al., 1984). Like Brooks' book, all three papers have survived the years extremely well; most of their insights are still as valid now as when they were first published.

This chapter draws upon these sources, plus the author's personal experience as designer or co-designer of three systems: Amoeba (Tanenbaum et al., 1990), MINIX (Tanenbaum and Woodhull, 1997), and Globe (Van Steen et al., 1999a).

Since no consensus exists among operating system designers about the best way to design an operating system, this chapter will thus be more personal, speculative, and undoubtedly more controversial than the previous ones.

12.1 THE NATURE OF THE DESIGN PROBLEM

Operating system design is more of an engineering project than an exact science. It is much harder to set clear goals and to meet them. Let us look at start with these points.

12.1.1 Goals

In order to design a successful operating system, the designers have to have a clear idea of what they want. Lack of a goal makes it very hard to make subsequent decisions. To make this point clearer, it is instructive to take a look at two programming languages PL/I and C. PL/I was designed by IBM in the 1960s because it was a nuisance to have to support both FORTRAN and COBOL, and embarrassing to have academics yapping in the background that Algol was better than both of them. So a committee was set up to produce a language that would be all things to all people: PL/I. It had a little bit of FORTRAN, a little bit of COBOL, and a little bit of Algol. It failed because it lacked any unifying vision. It was simply a collection of features at war with one another, and too cumbersome to be compiled efficiently, to boot.

Now consider C. It was designed by one person (Dennis Ritchie) for one purpose (system programming). It was a huge success, in no small part because Ritchie knew what he wanted and did not want. As a result, it is still in widespread use decades after its appearance. Having a clear vision of what you want is crucial.

What do operating system designers want? It obviously varies from system to system, being different for embedded systems than for server systems. However, for general-purpose operating systems four main items come to mind:

1. Define abstractions.
2. Provide primitive operations.
3. Ensure isolation.
4. Manage the hardware.

Each of these items will be discussed below.

The most important, but probably hardest task of an operating system is to define the right abstractions. Some of them, such as processes and files, have been around so long that they may seem obvious. Other ones, such as threads, are

newer, and are less mature. For example, if a multithread process that has one thread blocked waiting for keyboard input forks, is there a thread in the new process also waiting for keyboard input? Other abstractions relate to synchronization, signals, the memory model, modeling of I/O and many other areas.

Each of the abstractions can be instantiated in the form of concrete data structures. Users can create processes, files, semaphores, etc. The primitive operations manipulate these data structures. For example, users can read and write files. The primitive operations are implemented in the form of system calls. From the user's point of view, the heart of the operating system is formed by the abstractions and the operations on them available via the system calls.

Since multiple users can be logged into a computer at the same time, the operating system needs to provide mechanisms to keep them separated. One user may not interfere with another. The process concept is widely used to group resources together for protection purposes. Files and other data structures generally are protected as well. Making sure each user can perform only authorized operations on authorized data is a key goal of system design. However, users also want to share data and resources, so the isolation has to be selective and under user control. This makes it much harder.

Closely related to this point is the need to isolate failures. If some part of the system goes down, most commonly a user process, it should not be able to take the rest of the system down with it. The system design should make sure that the various parts are well isolated from one another. Ideally, parts of the operating system should also be isolated from one another to allow independent failures.

Finally, the operating system has to manage the hardware. In particular, it has to take care of all the low-level chips, such as interrupt controllers and bus controllers. It also has to provide a framework for allowing device drivers to manage the larger I/O devices, such as disks, printers, and the display.

12.1.2 Why is it Hard to Design an Operating System?

Moore's law says that computer hardware improves by a factor of 100 every decade. Nobody has a law saying that operating systems improve by a factor of 100 every decade. Or even get better at all. In fact, a case can be made that some of them are worse in key respects (such as reliability) than UNIX Version 7 was back in the 1970s.

Why? Inertia and the desire for backward compatibility often get much of the blame, and the failure to adhere to good design principles is also a culprit. But there is more to it. Operating systems are fundamentally different in certain ways from small application programs sold in computer stores for $49. Let us look at eight of the issues that make designing an operating system much harder than designing an application program.

First, operating systems have become extremely large programs. No one person can sit down at a PC and dash off a serious operating system in a few months.

All current versions of UNIX exceed 1 million lines of code; Windows 2000 is 29 million lines of code. No one person can understand even 1 million lines of code, let alone 29 million lines of code. When you have a product that none of the designers can hope to fully understand, it should be no surprise that the results are often far from optimal.

Operating systems are not the most complex systems around. Aircraft carriers are far more complicated, for example, but they partition into isolated subsystems better. The people designing the toilets on a aircraft carrier do not have to worry about the radar system. The two subsystems do not interact much. In an operating system, the file system often interacts with the memory system in unexpected and unforeseen ways.

Second, operating systems have to deal with concurrency. There are multiple users and multiple I/O devices all active at once. Managing concurrency is inherently much harder than managing a single sequential activity. Race conditions and deadlocks are just two of the problems that come up.

Third, operating systems have to deal with potentially hostile users—users who want to interfere with system operation or do things that are forbidden from doing, such as stealing another user's files. The operating system needs to take measures to prevent these users from behaving improperly. Word processing programs and photo editors do not have this problem.

Fourth, despite the fact that not all users trust each other, many users do want to share some of their information and resources with selected other users. The operating system has to make this possible, but in such a way that malicious users cannot interfere. Again, application programs do not face anything like this challenge.

Fifth, operating systems live for a very long time. UNIX has been around for a quarter of a century; Windows has been around for over a decade and shows no signs of vanishing. Consequently, the designers have to think about how hardware and applications may change in the distant future and how they should prepare for it. Systems that are locked too closely into one particular vision of the world usually die off.

Sixth, operating system designers really do not have a good idea of how their systems will be used, so they need to provide for considerable generality. Neither UNIX nor Windows was designed with email or Web browsers in mind, yet many computers running these systems do little else. Nobody tells a ship designer to build a ship without specifying whether they want a fishing vessel, a cruise ship, or a battleship. And even fewer change their minds after the product has arrived.

Seventh, modern operating systems are generally designed to be portable, meaning they have to run on multiple hardware platforms. They also have to support hundreds, even thousands of I/O devices, all of which are independently designed with no regard to one another. An example of where this diversity causes problems is the need for an operating system to run on both little-endian and big-endian machines. A second example was seen constantly under MS-DOS

when users attempted to install, say, a sound card and a modem that used the same I/O ports or interrupt request lines. Few programs other than operating systems have to deal with sorting out problems caused by conflicting pieces of hardware.

Eighth, and last in our list, is the frequent need to be backward compatible with some previous operating system. That system may have restrictions on word lengths, file names, or other aspects that the designers now regard as obsolete, but are stuck with. It is like converting a factory to produce next year's cars instead of this year's cars, but while continuing to produce this year's cars at full capacity.

12.2 INTERFACE DESIGN

It should be clear by now that writing a modern operating system is not easy. But where does one begin? Probably the best place to begin is to think about the interfaces it provides. An operating system provides a set of services, mostly data types (e.g., files) and operations on them (e.g., read). Together, these form the interface to its users. Note that in this context the users of the operating system are programmers who write code that use system calls, not people running application programs.

In addition to the main system call interface, most operating systems have additional interfaces. For example, some programmers need to write device drivers to insert into the operating system. These drivers see certain features and can make certain procedure calls. These features and calls also define and interface, but a very different one from one application programmers see. All of these interfaces must be carefully designed if the system is to succeed.

12.2.1 Guiding Principles

Are their any principles that can guide interface design? We believe there are. Briefly summarized, they are simplicity, completeness, and the ability to be implemented efficiently.

Principle 1: Simplicity

A simple interface is easier to understand and implement in a bug-free way. All system designers should memorize this famous quote from the pioneer French aviator and writer, Antoine de St. Exupéry:

> *Perfection is reached not when there is no longer anything to add, but when there is no longer anything to take away.*

This principle says that less is better than more, at least in the operating system itself. Another way to say this is the KISS principle: Keep It Simple, Stupid.

Principle 2: Completeness

Of course, the interface must make it possible to do everything that the users need to do, that is, it must be complete. This brings us to another famous quote, this one from Albert Einstein:

Everything should be as simple as possible, but no simpler.

In other words, the operating system should do exactly what is needed of it and no more. If users need to store data, it must provide some mechanism for storing data. If users need to communicate with each other, the operating system has to provide a communication mechanism, and so on. In his 1991 Turing Award lecture, Fernando Corbató, one of the designers of CTSS and MULTICS, combined the concepts of simplicity and completeness and said:.SP 0.25

First, it is important to emphasize the value of simplicity and elegance, for complexity has a way of compounding difficulties and as we have seen, creating mistakes. My definition of elegance is the achievement of a given functionality with a minimum of mechanism and a maximum of clarity.

The key idea here is *minimum of mechanism*. In other words, every feature, function, and system call should carry its own weight. It should do one thing and do it well. When a member of the design team proposes extending a system call or adding some new feature, the others should ask the question: "Would something awful happen if we left it out?" If the answer is: "No, but somebody might find this feature useful some day" put it in a user-level library, not in the operating system, even if it is slower that way. Not every feature has to be faster than a speeding bullet. The goal is to preserve what Corbató called minimum of mechanism.

Let us briefly consider two examples from my own experience: MINIX and Amoeba. For all intents and purposes, MINIX has three system calls: send, receive, and sendrec. The system is structured as a collection of processes, with the memory manager, the file system, and each device driver being a separate schedulable process. To a first approximation, all the kernel does is schedule processes and handle message passing between them. Consequently, only two system calls are needed: send, to send a message, and receive to receive one. The third call, sendrec is simply an optimization for efficiency reasons to allow a message to be sent and the reply to be requested with only one kernel trap. Everything else is done by requesting some other process (for example, the file system process or the disk driver) to do the work.

Amoeba is even simpler. It has only one system call: perform remote procedure call. This call sends a message and waits for a reply. It is essentially the same as MINIX' sendrec. Everything else is built on this one call.

Principle 3: Efficiency

The third guideline is efficiency of implementation. If a feature or system call cannot be implemented efficiently, it is probably not worth having. It should also be intuitively obviously to the programmer about how much a system call costs. For example, UNIX programmers expect the lseek system call to be cheaper than the read system call because the former just changes a pointer in memory while the latter performs disk I/O. If the intuitive costs are wrong, programmers will write inefficient programs.

12.2.2 Paradigms

Once the goals have been established, the design can begin. A good starting place is thinking about how the customers will view the system. One of the most important issues is how to make all the features of the system hang together well and present what is often called **architectural coherence**. In this regard, it is important to distinguish two kinds of operating system "customers." On the one hand, there are the *users*, who interact with application programs; on the other are the *programmers*, who write them. The former mostly deal with the GUI; the latter mostly deal with the system call interface. If the intention is to have a single GUI that pervades the complete system, such as in the Macintosh, the design should start there. If, on the other hand, the intention is to support many possible GUIs, such as in UNIX, the system call interface should be designed first. Doing the GUI first is essentially a top-down design. The issues are what features will it have, how will the user interact with it, and how should the system be designed to support it? For example, if most programs display icons on the screen and then wait for the user to click on one of them, this suggests an event-driven model for the GUI and probably also for the operating system. On the other hand, if the screen is mostly full of text windows, then a model in which processes read from the keyboard is probably better.

Doing the system call interface first is a bottom-up design. Here the issues are what kinds of features programmers in general need. Actually, not many special features are needed to support a GUI. For example, the UNIX windowing system, X, is just a big C program that does reads and writes on the keyboard, mouse, and screen. X was developed long after UNIX and did not require many changes to the operating system to get it to work. This experience validated that fact that UNIX was sufficiently complete.

User Interface Paradigms

For both the GUI-level interface and the system-call interface, the most important aspect is having a good paradigm (sometimes called a metaphor) to provide a way of looking at the interface. Many GUIs for desktop machines use the

WIMP paradigm that we discussed in Chap. 5. This paradigm uses point-and-click, point-and-double-click, dragging, and other idioms throughout the interface to provide an architectural coherence to the whole. Often there are additional requirements for programs, such as having a menu bar with FILE, EDIT, and other entries, each of which has certain well-known menu items. In this way, users who know one program can quickly learn another one.

However, the WIMP user interface is not the only possible one. Some palmtop computers use a stylized handwriting interface. Dedicated multimedia devices may use a VCR-like interface. And of course, voice input has a completely different paradigm. What is important is not so much the paradigm chosen, but the fact that there is a single overriding paradigm that unifies the entire user interface.

Whatever paradigm is chosen, it is important that all application programs use it. Consequently, the system designers need to provide libraries and tool kits to application developers that give them access to procedures that produce the uniform look-and-feel. User interface design is very important, but it is not the subject of this book, so we will now drop back down to the subject of the operating system interface.

Execution Paradigms

Architectural coherence is important at the user level, but equally important at the system call interface level. Here it is frequently useful to distinguish between the execution paradigm and the data paradigm, so we will do both, starting with the former.

Two execution paradigms are widespread: algorithmic and event driven. The **algorithmic paradigm** is based on the idea that a program is started to perform some function that it knows in advance or gets from its parameters. That function might be to compile a program, do the payroll, or fly an airplane to San Francisco. The basic logic is hardwired into the code with the program making system calls from time to time to get user input, obtain operating system services, etc. This approach is outlined in Fig. 12-1(a).

The other execution paradigm is the **event-driven paradigm** of Fig. 12-1(b). Here the program performs some kind of initialization, for example by displaying a certain screen, and then waiting for the operating system to tell it about the first event. The event is often a key being struck or a mouse movement. This design is useful for highly interactive programs.

Each of these ways of doing business engenders its own programming style. In the algorithmic paradigm, algorithms are central and the operating system is regarded as a service provider. In the event-driven paradigm, the operating system also provides services, but this role is overshadowed by its role as a coordinator of user activities and a generator of events that are consumed by processes.

```
main( )
{
     int ... ;

     init( );
     do_something( );
     read(...);
     do_something_else( );
     write(...);
     keep_going( );
     exit(0);
}
```

(a)

```
main( )
{
     mess_t msg;

     init( );
     while (get_message(&msg)) {
          switch (msg.type) {
               case 1: ... ;
               case 2: ... ;
               case 3: ... ;
          }
     }
}
```

(b)

Figure 12-1. (a) Algorithmic code. (b) Event-driven code.

Data Paradigms

The execution paradigm is not the only one exported by the operating system. An equally important one is the data paradigm. The key question here is how are system structures and devices presented to the programmer. In early FORTRAN batch systems, everything was modeled as a sequential magnetic tape. Card decks read in were treated as input tapes, card decks to be punched were treated as output tapes, and output for the printer was treated as an output tape. Disk files were also treated as tapes. Random access to a file was possible only by rewinding the tape corresponding to the file and reading it again.

The mapping was done using job control cards like these:

```
MOUNT(TAPE08, REEL781)
RUN(INPUT, MYDATA, OUTPUT, PUNCH, TAPE08)
```

The first card instructed the operator to go get tape reel 781 from the tape rack and mount it on tape drive 8. The second card instructed the operating system to run the just compiled FORTRAN program, mapping *INPUT* (meaning the card reader) to logical tape 1, disk file *MYDATA* to logical tape 2, the printer (called *OUTPUT*) to logical tape 3, the card punch (called *PUNCH*) to logical tape 4. and physical tape drive 8 to logical tape 5.

FORTRAN had a syntax for reading and writing logical tapes. By reading from logical tape 1, the program got card input. By writing to logical tape 3, output would later appear on the printer. By reading from logical tape 5, tape reel 781 could be read in, and so on. Note that the tape idea was just a paradigm to integrate the card reader, printer, punch, disk files, and tapes. In this example, only logical tape 5 was a physical tape; the rest were ordinary (spooled) disk files. It was a primitive paradigm, but it was a start in the right direction.

Later came UNIX, which goes much further using the model of "everything is a file." Using this paradigm, all I/O devices are treated as files and can be opened and manipulated as ordinary files. The C statements

```
fd1 = open("file1", O_RDWR);
fd2 = open("/dev/tty", O_RDWR)'
```

open a true disk file and the user's terminal. Subsequent statements can use *fd1* and *fd2* to read and write them, respectively. From that point on, there is no difference between accessing the file and accessing the terminal, except that seeks on the terminal are not allowed.

Not only does UNIX unify files and I/O devices, but it also allows other processes to be accessed over pipes as files. Furthermore, when mapped files are supported, a process can get at its own virtual memory as though it were a file. Finally, in versions of UNIX that support the */proc* file system, the C statement

```
fd3 = open("/proc/501", O_RDWR);
```

allows the process to (try to) access process 501's memory for reading and writing using file descriptor *fd3*, something useful for, say, a debugger.

Windows 2000 goes further still and tries to make everything look like an object. Once a process has acquired a valid handle to a file, process, semaphore, mailbox, or other kernel object, it can perform operations on it. This paradigm is even more general than that of UNIX and much more general than that of FORTRAN.

Unifying paradigms occur in other contexts as well. One of them is worth mentioning here: the Web. The paradigm behind the Web is that cyberspace is full of documents, each of which has a URL. By typing in a URL or clicking on an entry backed by a URL, you get the document. In reality, many "documents" are not documents at all, but are generated by a program or shell script when a request comes in. For example, when a user asks an online store for a list of CDs by a particular artist, the document is generated on-the-fly by a program; it certainly did not exist before the query was made.

We have now seen four cases: namely, everything is a tape, file, object, or document. In all four cases, the intention is to unify data, devices, and other resources to make them easier to deal with. Every operating system should have such a unifying data paradigm.

12.2.3 The System Call Interface

If one believes in Corbató's dictum of minimal mechanism, then the operating system should provide as few system calls as it can get away with, and each one should be as simple as possible (but no simpler). A unifying data paradigm can play a major role in helping here. For example, if files, processes, I/O devices, and much more all look like files or objects, then they can all be read with a single

read system call. Otherwise it may be necessary to have separate calls for read_file, read_proc, and read_tty, among others.

In some cases, system calls may appear to need several variants, but it is often better practice to have one system call that handles the general case, with different library procedures to hide this fact from the programmers. For example, UNIX has a system call for overlaying a process virtual address space, exec. The most general call is

```
exec(name, argp, envp);
```

which loads the executable file *name* and gives it arguments pointed to by *argp* and environment variables pointed to by *envp*. Sometimes it is convenient to list the arguments explicitly, so the library contains procedures that are called as follows:

```
execl(name, arg0, arg1, ..., argn, 0);
execle(name, arg0, arg1, ..., argn, envp);
```

All these procedures do is stick the arguments in an array and then call exec to do the work. This arrangement is the best of both worlds: a single straightforward system call keeps the operating system simple, yet the programmer gets the convenience of various ways to call exec.

Of course, trying to have one call to handle every possible case can easily get out of hand. In UNIX creating a process requires two calls: fork followed by exec. The former has no parameters; the latter has three parameters. In contrast, the Win32 API call for creating a process, CreateProcess, has 10 parameters, one of which is a pointer to a structure with an additional 18 parameters.

A long time ago, someone should have asked the question: "Would something awful happen if we left some of these out?" The truthful answer would have been: "In some cases programmers might have to do more work to achieve a particular effect, but the net result would have been a simpler, smaller, and more reliable operating system." Of course, the person proposing the 10+ 18 parameter version might have added: "But users like all these features." The rejoinder to that might have been: "They like systems that use little memory and never crash even more." Trade-offs between more functionality at the cost of more memory are at least visible and can be given a price tag (since the price of memory is known). However, it is hard to estimate the additional crashes per year some feature will add and whether the users would make the same choice if they knew the hidden price. This effect can be summarized in Tanenbaum's first law of software:

Adding more code adds more bugs

Adding more features adds more code and thus adds more bugs. Programmers who believe adding new features does not add new bugs are either new to computers or believe the tooth fairy is out there watching over them.

Simplicity is not the only issue that comes out when designing system calls. An important consideration is Lampson's (1984) slogan:

Don't hide power.

If the hardware has an extremely efficient way of doing something, it should be exposed to the programmers in a simple way and not buried inside some other abstraction. The purpose of abstractions is to hide undesirable properties, not hide desirable ones. For example, suppose that the hardware has a special way to move large bitmaps around the screen (i.e., the video RAM) at high speed. It would be justified to have a new system call to get at this mechanism, rather than just providing ways to read video RAM into main memory and write it back again. The new call should just move bits and nothing else. If a system call is fast, users can always build more convenient interfaces on top of it. If it is slow, nobody will use it.

Another design issue is connection-oriented versus connectionless calls. The standard UNIX and Win32 system calls for reading a file are connection-oriented. First you open a file, then you read it, finally you close it. Some remote file access protocols are also connection-oriented. For example, to use FTP, the user first logs in to the remote machine, reads the files, and then logs out.

On the other hand, some remote file access protocols are connectionless. NFS is connectionless, as we saw in Chap. 10. Each NFS call stands on its own, so files are not opened before reading or writing. And, of course, they do not have to be closed afterward, either. The Web is also connectionless: to read a Web page you just ask for it; there is no advance setup required (a TCP connection *is* required, but this is at a lower level of protocol; the HTTP protocol for accessing the Web itself is connectionless).

The trade-off between any connection-oriented mechanism and a connectionless one is the additional work required to set up the mechanism (e.g., open the file), and the gain from not having to do it on (possibly many) subsequent calls. For file I/O on a single machine, where the setup cost is low, probably the standard way (first open, then use) is the best way. For remote file systems, a case can be made both ways.

Another issue relating to the system call interface is its visibility. The list of POSIX-mandated system calls is easy to find. All UNIX systems support these, as well as a small number of other calls, but the complete list is always public. In contrast, Microsoft has never made the list of Windows 2000 system calls public. Instead the Win32 API and other APIs have been made public, but these contain vast numbers of library calls (over 13,000 for Windows 2000), but only a small number are true system calls. The argument for making all the system calls public is that it lets programmers know what is cheap (functions performed in user space) and what is expensive (kernel calls). The argument for not making them public is that it gives the implementers the flexibility of changing the actual underlying system calls to make them better without breaking user programs.

12.3 IMPLEMENTATION

Turning away from the user and system call interfaces, let us now take a look at how to implement an operating system. In the next eight sections we will examine some general conceptual issues relating to implementation strategies. After that we will look at some low-level techniques that are often helpful.

12.3.1 System Structure

Probably the first decision the implementers have to make is what the system structure should be. We examined the main possibilities in Sec. 1.7, but will review them here. An unstructured monolithic design is really not a good idea, except maybe for a tiny operating system in, say, a refrigerator, but even there it is arguable.

Layered Systems

A reasonable approach that has been well established over the years is a layered system. Dijkstra's THE system (Fig. 1-25) was the first layered operating system. UNIX (Fig. 10-3) and Windows 2000 (Fig. 11-7) also have a layered structure, but the layering in both of them is more a way of trying to describe the system than a real guiding principle that was used in building the system.

For a new system, designers choosing to go this route should *first* very carefully choose the layers and define the functionality of each one. The bottom layer should always try to hide the worst idiosyncracies of the hardware, as the HAL does in Fig. 11-7. Probably the next layer should handle interrupts, context switching, and the MMU, so above this level, the code is mostly machine independent. Above this, different designers will have different tastes (and biases). One possibility is to have layer 3 manage threads, including scheduling and interthread synchronization, as shown in Fig. 12-2. The idea here is that starting at layer 4 we have proper threads that are scheduled normally and synchronize using a standard mechanism (e.g., mutexes).

In layer 4 we might find the device drivers, each one running as a separate thread, with its own state, program counter, registers, etc., possibly (but not necessarily) within the kernel address space. Such a design can greatly simplify the I/O structure because when an interrupt occurs, it can be converted into an unlock on a mutex and a call to the scheduler to (potentially) schedule the newly readied thread that was blocked on the mutex. MINIX uses this approach, but in UNIX, Linux, and Windows 2000, the interrupt handlers run in a kind of no-man's land, rather than as proper threads that can be scheduled, suspended, etc. Since a huge amount of the complexity of any operating system is in the I/O, any technique for making it more tractable and encapsulated is worth considering.

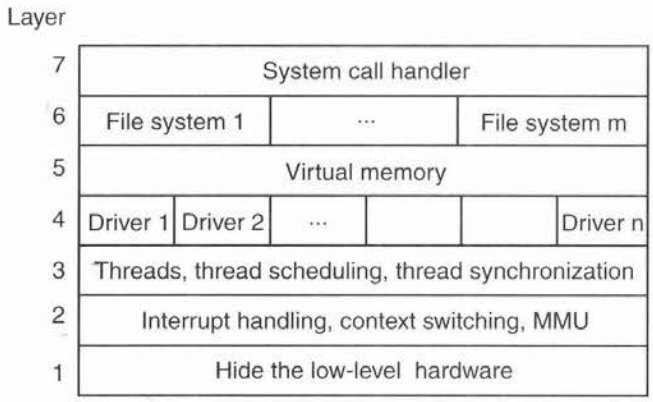


Figure 12-2. One possible design for a modern layered operating system.

Above layer 4, we would expect to find virtual memory, one or more file systems, and the system call handlers. If the virtual memory is at a lower level than the file systems, then the block cache can be paged out, allowing the virtual memory manager to dynamically determine how the real memory should be divided among user pages and kernel pages, including the cache. Windows 2000 works this way.

Exokernels

While layering has its supporters among system designers, there is also another camp that has precisely the opposite view (Engler et al., 1995). Their view is based on the **end-to-end argument** (Saltzer et al., 1984). This concept says that if something has to be done by the user program itself, it is wasteful to do it in a lower layer as well.

Consider an application of that principle to remote file access. If a system is worried about data being corrupted in transit, it should arrange for each file to be checksummed at the time it is written and the checksum stored along with the file. When a file is transferred over a network from the source disk to the destination process, the checksum is transferred, too, and also recomputed at the receiving end. If the two disagree, the file is discarded and transferred again.

This check is more accurate than using a reliable network protocol since it also catches disk errors, memory errors, software errors in the routers, and other errors besides bit transmission errors. The end-to-end argument says that using a reliable network protocol is then not necessary, since the end point (the receiving process) has enough information to verify the correctness of the file itself. The only reason for using a reliable network protocol in this view is for efficiency, that is, catching and repairing transmission errors earlier.

The end-to-end argument can be extended to almost all of the operating system. It argues for not having the operating system do anything that the user