# 1 INTRODUCTION

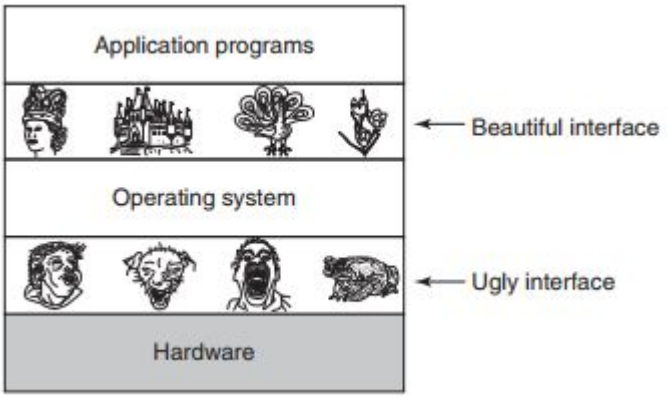
## *1.1 WHAT IS AN OPERATING SYSTEM?*

The operating systems perform two essentially unrelated functions:

- providing application programmers (and application programs, naturally) a clean abstract set of resources instead of the messy hardware ones
- managing these hardware resources

### *1.1.1 The Operating System as an Extended Machine*

One of the major tasks of the operating system is to hide the hardware and present programs (and their programmers) with nice, clean, elegant, consistent, abstractions to work with instead. Operating systems turn the ugly into the beautiful.




### *1.1.2 The Operating System as a Resource Manager*

The job of the operating system is to provide for an orderly and controlled allocation of the processors, memories, and I/O devices among the various programs wanting them. In short, this view of the operating system holds that its primary task is to keep track of which programs are using which resource, to grant resource requests, to account for usage, and to mediate conflicting requests from different programs and users.

Resource management includes multiplexing (sharing) resources in two different ways:

- Time: Determining how the resource is time multiplexed—who goes next and for how long—is the task of the operating system.
- Space: Instead of the customers taking turns, each one gets part of the resource. For example, main memory is normally divided up among several running programs, so each one can be resident at the same time (for example, in order to take turns using the CPU).

## *1.5 OPERATING SYSTEM CONCEPTS*

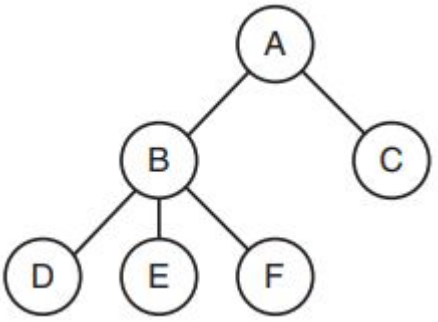
Most operating systems provide certain basic concepts and abstractions such as processes, address spaces, and files that are central to understanding them.

### *1.5.1 Processes*

A key concept in all operating systems is the process. A process is basically a program in execution. Associated with each process is its **address space**, a list of memory locations from 0 to some maximum, which the process can read and write. The address space contains the executable program, the program's data, and its stack. Also associated with each process is a set of resources, commonly including registers (including the program counter and stack pointer), a list of open files, outstanding alarms, lists of related processes, and all the other information needed to run the program. A process is fundamentally a container that holds all the information needed to run a program.

All the information about each process, other than the contents of its own address space, is stored in an operating system table called the **process table**, which is an array of structures, one for each process currently in existence.

A (suspended) process consists of its address space, usually called the **core image** (in honor of the magnetic core memories used in days of yore), and its process table entry, which contains the contents of its registers and many other items needed to restart the process later.




If a process can create one or more other processes (referred to as **child processes**) and these processes in turn can create child processes, we quickly arrive at the process tree structure. Communication between child and parent processes is called **interprocess communication**.

An **alarm signal** causes the process to temporarily suspend whatever it was doing, save its registers on the stack, and start running a special signal-handling procedure, for example, to retransmit a presumably lost message. When the signal handler is done, the running process is restarted in the state it was in just before the signal.
Signals are the software analog of hardware **interrupts** and can be generated by a variety of causes in addition to timers expiring.

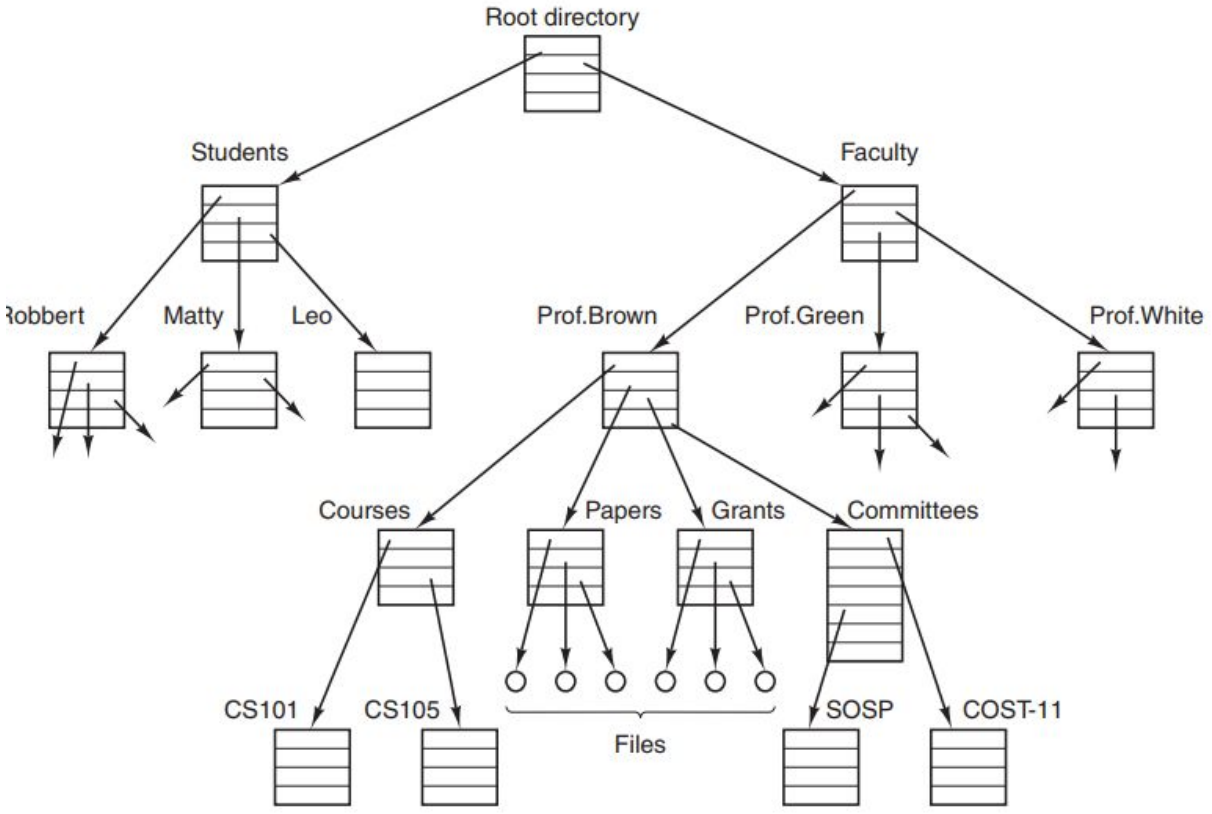
Each person authorized to use a system is assigned a **UID** (User IDentification) by the system administrator. Every process started has the UID of the person who started it. A child process has the same UID as its parent. Users can be members of groups, each of which has a **GID** (Group IDentification).
One UID, called the **superuser** (in UNIX), or **Administrator** (in Windows), has special power and may override many of the protection rules.

### *1.5.2 Address Spaces*

The operating system is concerned with managing and protecting the computer's main memory. A different, but equally important, memory-related issue is managing the address space of the processes.
Each process has some set of addresses it can use, typically running from 0 up to some maximum. In the simplest case, the maximum amount of address space a process has is less than the main memory. In this way, a process can fill up its address space and there will be enough room in main memory to hold it all.
What happens if a process has more address space than the computer's main memory and the process wants to use it all? A technique called virtual memory exists, as mentioned earlier, in which the operating system keeps part of the address space in main memory and part on disk and shuttles pieces back and forth between them as needed.
The operating system creates the abstraction of an address space as the set of addresses a process may reference. The address space is decoupled from the machine's physical memory and may be either larger or smaller than the physical memory.

### *1.5.3 Files*

To provide a place to keep files, most PC operating systems have the concept of a **directory** as a way of grouping files together.




Every file within the directory hierarchy can be specified by giving its **path name** from the top of the directory hierarchy, the **root directory**.

Absolute path name: /Faculty/Prof.Brown/Courses/CS101. (begin with slash at root)

At every instant, each process has a **current working directory**, in which path names not beginning with a slash are looked for.

current directory path name: Courses/CS101 (current directory: /Faculty/Prof.Brown/)

Before a file can be read or written, it must be opened, at which time the permissions are checked. If the access is permitted, the system returns a small integer called a **file descriptor** to use in subsequent operations.

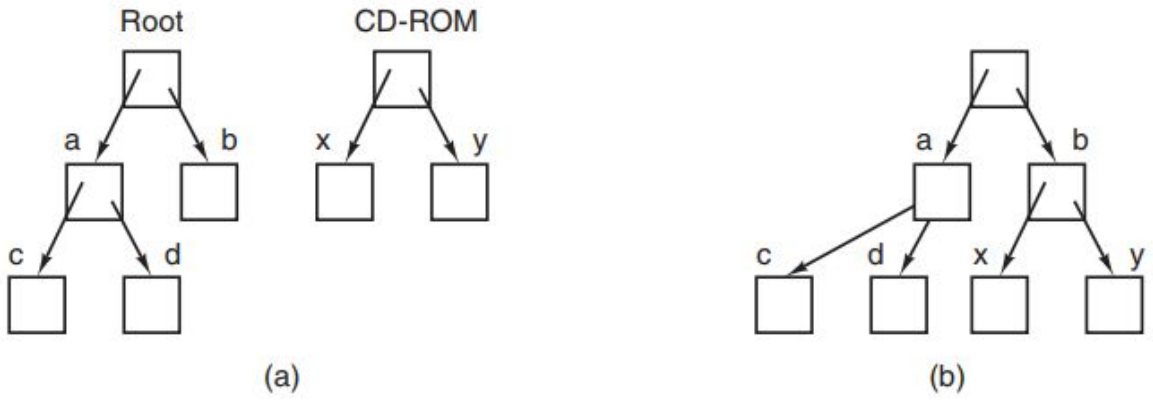



**Figure 1-15.** (a) Before mounting, the files on the CD-ROM are not accessible. (b) After mounting, they are part of the file hierarchy.

Another important concept in UNIX is the special file. Special files are provided in order to make I/O devices look like files. That way, they can be read and written using the same system calls as are used for reading and writing files. Two kinds of special files exist: **block special files** and **character special files**.

Block special files are used to model devices that consist of a collection of randomly addressable blocks, such as disks.
Character special files are used to model printers, modems, and other devices that accept or output a character stream.

A **pipe** is a sort of pseudofile that can be used to connect two processes.
If processes A and B wish to talk using a pipe, they must set it up in advance. When process A wants to send data to process B, it writes on the pipe as though it were an output file. In fact, the implementation of a pipe is very much like that of a file. Process B can read the data by reading from the pipe as though it were an input file.

### *1.5.4 Input/Output*
Every operating system has an I/O subsystem for managing its I/O devices.

1.5.5 Protection
Computers contain large amounts of information that users often want to protect and keep confidential. It is up to the operating system to manage the system security so that files, for example, are accessible only to authorized users.

Files in UNIX are protected by assigning each one a 9-bit binary protection code. The protection code consists of three 3-bit fields. Each field has a bit for read access, a bit for write access, and a bit for execute access. These 3 bits are known as the **rwx bits**.

the protection code rwxr-x--x means that the owner can read, write, or execute the file, other group members can read or execute (but not write) the file, and everyone else can execute (but not read or write) the file.
For a directory, x indicates search permission.

### *1.5.6 The Shell*

When any user logs in, a shell is started up. The shell has the terminal as standard input and standard output. It starts out by typing the prompt, a character such as a dollar sign, which tells the user that the shell is waiting to accept a command. If the user now types

date

for example, the shell creates a child process and runs the date program as the child.

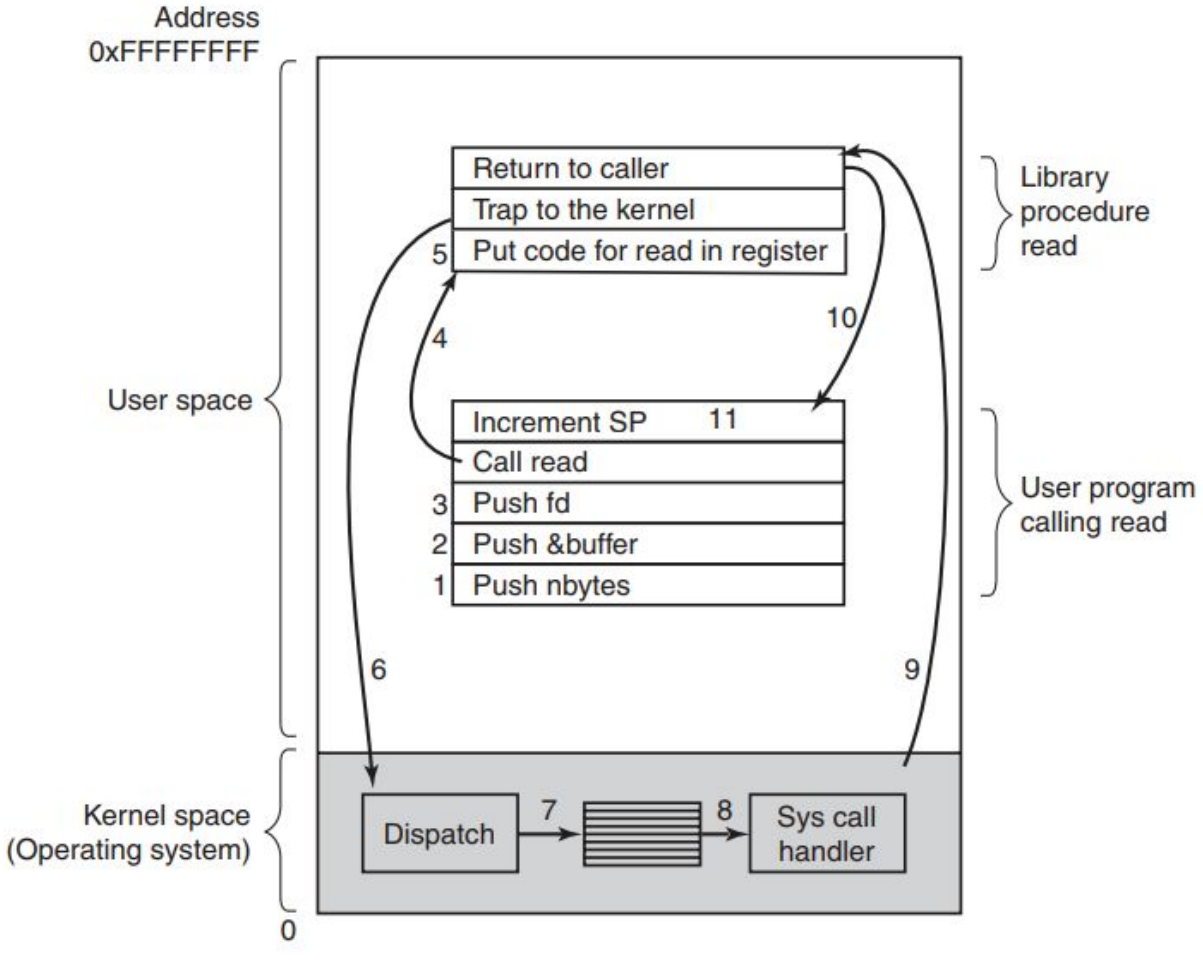
## *1.6 SYSTEM CALLS*

Any single-CPU computer can execute only one instruction at a time. If a process is running a user program in user mode and needs a system service, such as reading data from a file, it has to execute a trap instruction to transfer control to the operating system. The operating system then figures out what the calling process wants by inspecting the parameters. Then it carries out the system call and returns control to the instruction following the system call.

Making a system call is like making a special kind of procedure call, only system calls enter the kernel and procedure calls do not.

count = read(fd, buffer, nbytes);

The system call (and the library procedure) return the number of bytes actually read in count.



1. nbytes is pushed onto the stack
2. &buffer is pushed onto the stack
3. fd is pushed onto the stack
4. Then comes the actual call to the library procedure
5. The library procedure, possibly written in assembly language, typically puts the system-call number in a place where the operating system expects it, such as a register.
6. Then it executes a TRAP instruction to switch from user mode to kernel mode and start execution at a fixed address within the kernel
7. The kernel code that starts following the TRAP examines the system-call number and then dispatches to the correct system-call handler, usually via a table of pointers to system-call handlers indexed on system-call number
8. At that point the system-call handler runs
9. Once it has completed its work, control may be returned to the user-space library procedure at the instruction following the TRAP instruction
10. This procedure then returns to the user program in the usual way procedure calls return
11. To finish the job, the user program has to clean up the stack, as it does after any procedure call

In step 9 above, we said ‘‘may be returned to the user-space library procedure’’ for good reason. The system call may block the caller, preventing it from continuing. For example, if it is trying to read from the keyboard and nothing has been typed yet, the caller has to be blocked. In this case, the operating system will look around to see if some other process can be run next. Later, when the desired input is available, this process will get the attention of the system and run steps 9–11.

| Call | Description |
|---|---|
| pid = fork( ) | Create a child process identical to the parent |
| pid = waitpid(pid, &statloc, options) | Wait for a child to terminate |
| s = execve(name, argv, environp) | Replace a process' core image |
| exit(status) | Terminate process execution and return status |

**File management**

| Call | Description |
|---|---|
| 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 |

**Directory- and file-system management**

| Call | Description |
|---|---|
| s = mkdir(name, mode) | Create a new directory |
| s = rmdir(name) | Remove an empty directory |
| s = link(name1, name2) | Create a new entry, name2, pointing to name1 |
| s = unlink(name) | Remove a directory entry |
| s = mount(special, name, flag) | Mount a file system |
| s = umount(special) | Unmount a file system |

**Miscellaneous**

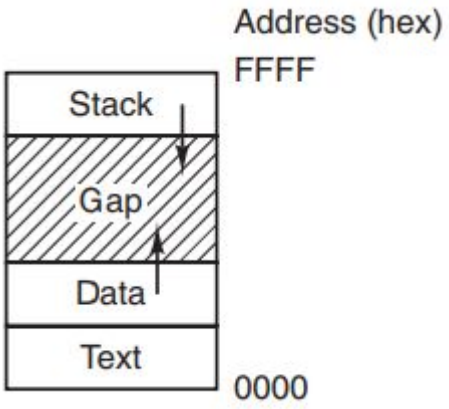
| Call | Description |
|---|---|
| s = chdir(dirname) | Change the working directory |
| s = chmod(name, mode) | Change a file's protection bits |
| s = kill(pid, signal) | Send a signal to a process |
| seconds = time(&seconds) | Get the elapsed time since Jan. 1, 1970 |

### *1.6.1 System Calls for Process Management*

**Fork** is a good place to start the discussion. Fork is the only way to create a new process in POSIX. It creates an exact duplicate of the original process, including all the file descriptors, registers—everything. After the fork, the original process and the copy (the parent and child) go their separate ways.All the variables have identical values at the time of the fork, but since the parent's data are copied to create the child, subsequent changes in one of them do not affect the other one. The fork call returns a value, which is zero in the child and equal to the child's **PID** (Process IDentifier) in the parent. Using the returned PID, the two processes can see which one is the parent process and which one is the child process.

**Waitpid** can wait for a specific child, or for any old child by setting the first parameter to −1. When waitpid completes, the address pointed to by the second parameter, statloc, will be set to the child process' exit status

cp file1 file2
main(argc, argv, envp)
**argc** is a count of the number of items on the command line, including the program name.
**argv** is a pointer to an array. Element i of that array is a pointer to the ith string on the command line. In our example, argv[0] would point to the string ‘‘cp’’, argv[1] would point to the string ‘‘file1’’, and argv[2] would point to the string ‘‘file2’’.
**envp** is a pointer to the environment, an array of strings containing assignments of the form name = value used to pass information such as the terminal type and home directory name to programs.




Processes in UNIX have their memory divided up into three segments: the text segment (i.e., the program code), the data segment (i.e., the variables), and the stack segment. The data segment grows upward and the stack grows downward. Between them is a gap of unused address space.

1.6.2 System Calls for File Management
To read or write a file, it must first be opened. This call specifies the file name to be opened, either as an absolute path name or relative to the working directory, as well as a code of **O RDONLY**, **O WRONLY**, or **O RDWR**, meaning open for reading, writing, or both. To create a new file, the **O CREAT** parameter is used. The file descriptor returned can then 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 open.

The **lseek** call changes the value of the position pointer, so that subsequent calls to read or write can begin anywhere in the file.
Lseek has three parameters: the first is the file descriptor for the file, the second is a file position, and the third tells whether the file position is relative to the beginning of the file, the current position, or the end of the file. The value returned by lseek is the absolute position in the file (in bytes) after changing the pointer.

### *1.6.3 System Calls for Directory Management*

The first two calls, **mkdir** and **rmdir**, create and remove empty directories, respectively. The next call is **link**. Its purpose is to allow the same file to appear under two or more names, often in different directories.

Every file in UNIX has a unique number, its **i-number**, that identifies it. This i-number is an index into a table of i-nodes, one per file, telling who owns the file, where its disk blocks are, and so on. A directory is simply a file containing a set of (i-number, ASCII name) pairs. What link does is simply create a brand new directory entry with a (possibly new) name, using the i-number of an existing file.

The **mount** system call allows two file systems to be merged into one.

### *1.6.4 Miscellaneous System Calls*

The **chdir** call changes the current working directory.
**chdir**("/usr/ast/test");

The **chmod** system call makes it possible to change the mode of a file.
**chmod**("file", 0644);

The **kill** system call is the way users and user processes send signals. If a process is prepared to catch a particular signal, then when it arrives, a signal handler is run. If the process is not prepared to handle a signal, then its arrival kills the process (hence the name of the call).

**time** just returns the current time in seconds, with 0 corresponding to Jan. 1, 1970 at midnight.

## 1.7 OPERATING SYSTEM STRUCTURE

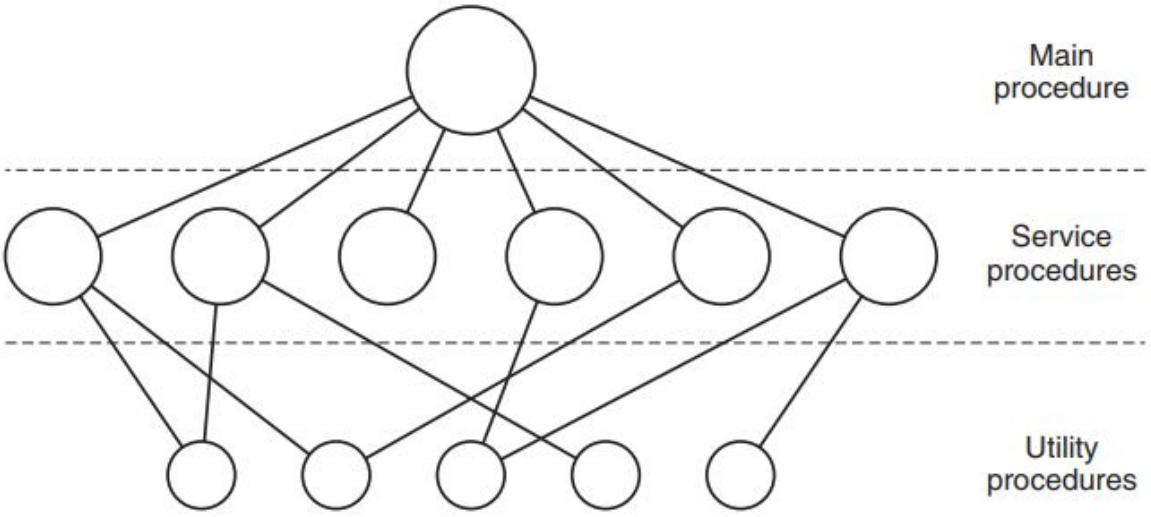
The six designs we will discuss here are monolithic systems, layered systems, microkernels, client-server systems, virtual machines, and exokernels.

### 1.7.1 Monolithic Systems

By far the most common organization, in the monolithic approach the entire operating system runs as a single program in kernel mode.
The operating system is written as a collection of procedures, linked together into a single large executable binary program.
Being able to call any procedure you want is very efficient, but having thousands of procedures that can call each other without restriction may also lead to a system that is unwieldy and difficult to understand. Also, a crash in any of these procedures will take down the entire operating system.

This organization suggests a basic structure for the operating system:
1. A main program that invokes the requested service procedure.
2. A set of service procedures that carry out the system calls.
3. A set of utility procedures that help the service procedures.

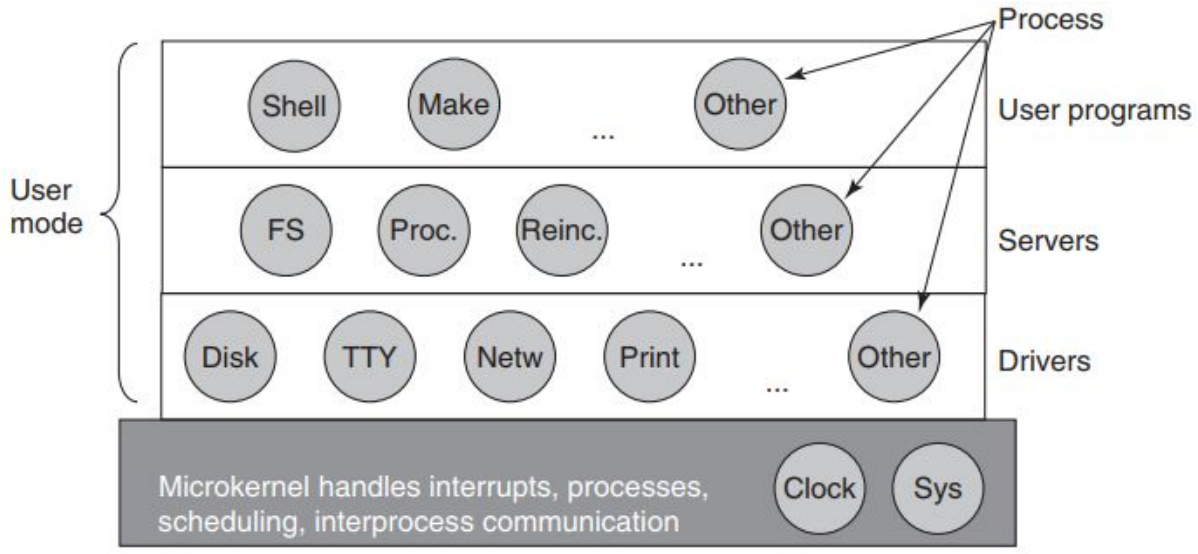
In addition to the core operating system that is loaded when the computer is booted, many operating systems support loadable extensions, such as I/O device drivers and file systems. in UNIX they are called **shared libraries**. In Windows they are called **DLLs** (Dynamic-Link Libraries).




### *1.7.3 Microkernels*

Microkernels: putting as little as possible in kernel mode because bugs in the kernel can bring down the system instantly.

The basic idea behind the microkernel design is to achieve high reliability by splitting the operating system up into small, well-defined modules, only one of which—the microkernel—runs in kernel mode and the rest run as relatively powerless ordinary user processes.

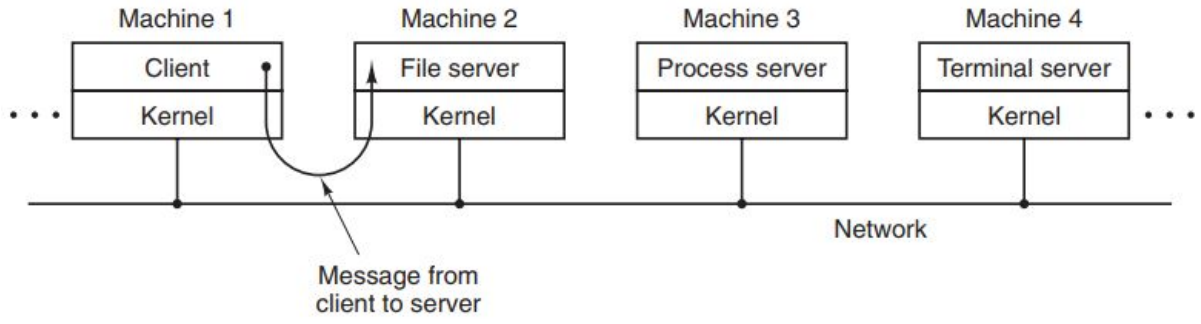



**Reincarnation server,** whose job is to check if the other servers and drivers are functioning

An idea somewhat related to having a minimal kernel is to put the **mechanism** for doing something in the kernel but not the **policy.**

### *1.7.4 Client-Server Model*

A slight variation of the microkernel idea is to distinguish two classes of processes, the servers, each of which provides some service, and the clients, which use these services. This model is known as the client-server model.



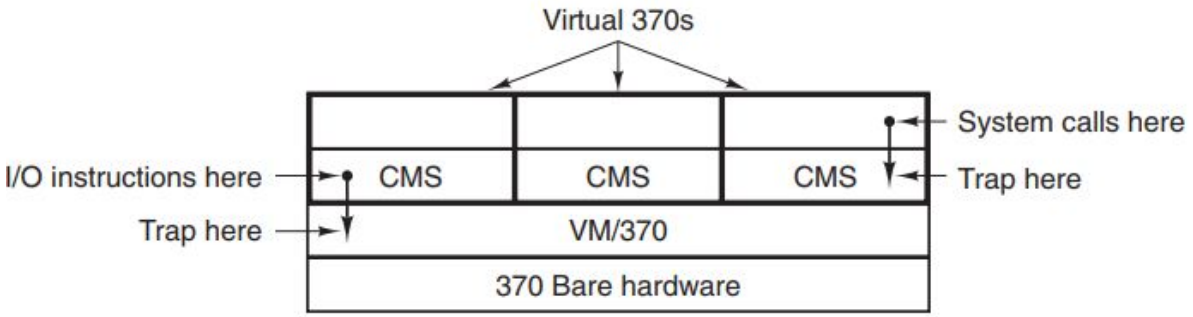

### *1.7.5 Virtual Machines*

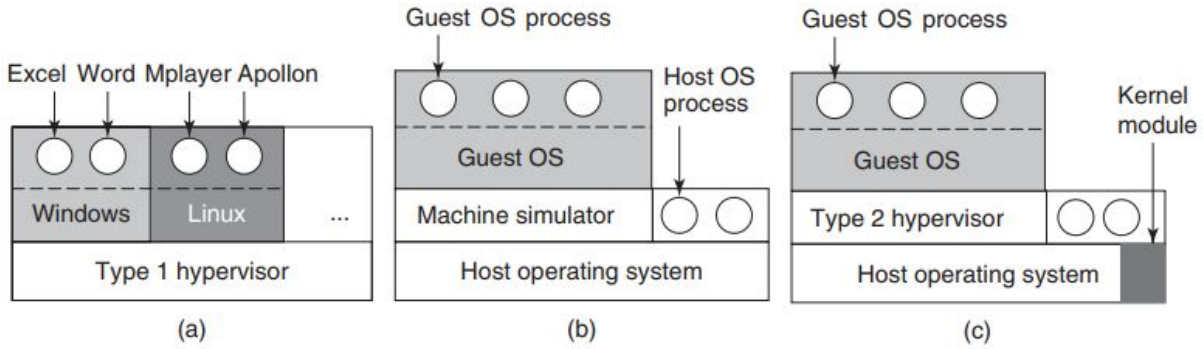
**VM/370**: a timesharing system provides (1) multiprogramming and (2) an extended machine with a more convenient interface than the bare hardware. The essence of VM/370 is to completely separate these two functions.

The heart of the system, known as the **virtual machine monitor**, runs on the bare hardware and does the multiprogramming, providing not one, but several virtual machines to the next layer up.

these virtual machines are not extended machines, with files and other nice features. They are exact copies of the bare hardware, including kernel/user mode, I/O, interrupts, and everything else the real machine has.

Because each virtual machine is identical to the true hardware, each one can run any operating system that will run directly on the bare hardware






Some of these early research projects improved the performance over interpreters like Bochs by translating blocks of code on the fly, storing them in an internal cache, and then reusing them if they were executed again. This improved the performance considerably, and led to what we will call **machine simulators**.
This technique, known as **binary translation**, helped improve matters, the resulting systems, while good enough to publish papers about in academic conferences, were still not fast enough to use in commercial environments where performance matters a lot. **(B)**

The next step in improving performance was to add a kernel module to do some of the heavy lifting. They are called **type 2 hypervisors.**

The real distinction between a type 1 hypervisor and a type 2 hypervisor is that a type 2 makes uses of a host operating system and its file system to create processes, store files, and so on. A type 1 hypervisor has no underlying support and must perform all these functions itself.

Another area where virtual machines are used, but in a somewhat different way, is for running Java programs. The advantage of this approach is that the JVM code can be shipped over the Internet to any computer that has a JVM interpreter and run there.

### *1.7.6 Exokernels*

Rather than cloning the actual machine, as is done with virtual machines, another strategy is partitioning it, in other words, giving each user a subset of the resources. Thus one virtual machine might get disk blocks 0 to 1023, the next one might get blocks 1024 to 2047, and so on.
At the bottom layer, running in kernel mode, is a program called the **exokernel.**
Its job is to allocate resources to virtual machines and then check attempts to use them to make sure no machine is trying to use somebody else's resources.

# 2 PROCESSES AND THREADS

The most central concept in any operating system is the process: an abstraction of a running program.

## *2.1 PROCESSES*

In any multiprogramming system, the CPU switches from process to process quickly, running each for tens or hundreds of milliseconds. While, strictly speaking, at any one instant the CPU is running only one process, in the course of 1 second it may work on several of them, giving the illusion of parallelism. Sometimes people speak of **pseudoparallelism** in this context, to contrast it with the true hardware parallelism of **multiprocessor** systems (which have two or more CPUs sharing the same physical memory).

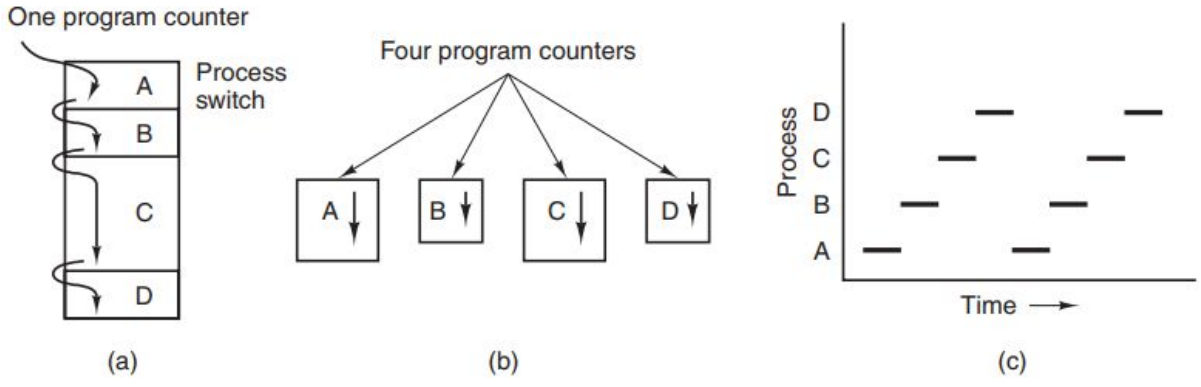
### *2.1.1 The Process Model*

In this model, all the runnable software on the computer, sometimes including the operating system, is organized into a number of sequential processes, or just processes for short.

**process:** an instance of an executing program, including the current values of the program counter, registers, and variables.

Conceptually, each process has its own virtual CPU. In reality, of course, the real CPU switches back and forth from process to process. This rapid switching back and forth is called **multiprogramming**.

Each process has its own flow of control (i.e., its own logical program counter). There is only one physical program counter. So when each process runs, its logical program counter is loaded into the real program counter. When it is finished (for the time being), the physical program counter is saved in the process' stored logical program counter in memory.




(a) Multiprogramming four programmes. (b) Conceptual model of four independent, sequential processes. (c) Only one program is active at once.

The rate at which a process performs its computation is not uniform. Thus, processes must not be programmed with built-in assumptions about timing.

When a process has critical real-time requirements, like in a video, special measures must be taken to ensure that they do occur.

The key idea here is that a process is an activity of some kind. It has a program, input, output, and a state. A single processor may be shared among several processes, with some scheduling algorithm being accustomed to determine when to stop work on one process and service a different one.

It is worth noting that if a program is running twice, it counts as two processes.

### *2.1.2 Process Creation*

In general-purpose systems some way is needed to create and terminate processes as needed during operation.

Four principal events cause processes to be created:

1. System initialization.
2. Execution of a process-creation system call by a running process.
3. A user request to create a new process.
4. Initiation of a batch job.

When an operating system is booted, typically numerous processes are created.

Processes that stay in the background to handle some activity such as email, Web pages, news, printing, and so on are called **daemons.**

Often a running process will issue system calls to create one or more new processes to help it do its job.

In interactive systems, users can start a program by typing a command or (double) clicking on an icon. Taking either of these actions starts a new process and runs the selected program in it. In command-based UNIX systems running X, the new process takes over the window in which it was started.

The last situation in which processes are created applies only to the batch systems found on large mainframes.

in all these cases, a new process is created by having an existing process execute a process creation system call. That process may be a running user process, a system process invoked from the keyboard or mouse, or a batch-manager process. What that process does is execute a system call to create the new process. This system call tells the operating system to create a new process and indicates, directly or indirectly, which program to run in it.

In UNIX, there is only one system call to create a new process: **fork**. This call creates an exact clone of the calling process. After the fork, the two processes, the parent and the child, have the same memory image, the same environment strings, and the same open files.

Usually, the child process then executes **execve** or a similar system call to change its memory image and run a new program.

The reason for this two-step process is to allow the child to manipulate its file descriptors after the fork but before the execve in order to accomplish redirection of standard input, standard output, and standard error.

In both UNIX and Windows systems, after a process is created, the parent and child have their own distinct address spaces. If either process changes a word in its address space, the change is not visible to the other process. In UNIX, the child's initial address space is a copy of the parent's, but there are definitely two distinct address spaces involved; no writable memory is shared.

The child may share all of the parent's memory, but in that case the memory is shared copy-on-write, which means that whenever either of the two wants to modify part of the memory, that chunk of memory is explicitly copied first to make sure the modification occurs in a private memory area.

### *2.1.3 Process Termination*

Processes can be terminated by:

1. Normal exit (voluntary).
2. Error exit (voluntary).
3. Fatal error (involuntary).
4. Killed by another process (involuntary).

Most processes terminate because they have done their work. When a compiler has compiled the program given to it, the compiler executes a system call to tell the operating system that it is finished. This call is exit in UNIX and ExitProcess in Windows.

Screen-oriented programs also support voluntary termination. Word processors, Internet browsers, and similar programs always have an icon or menu item that the user can click to tell the process to remove any temporary files it has open and then terminate.

The second reason for termination is that the process discovers a fatal error.

Screen-oriented interactive processes generally do not exit when given bad parameters. Instead they pop up a dialog box and ask the user to try again.

The third reason for termination is an error caused by the process, often due to a program bug. Examples include executing an illegal instruction, referencing nonexistent memory, or dividing by zero.

In some systems (e.g., UNIX), a process can tell the operating system that it wishes to handle certain errors itself, in which case the process is signaled (interrupted) instead of terminated when one of the errors occurs.

The fourth reason a process might terminate is that the process executes a system call telling the operating system to kill some other process. In UNIX this call is kill. The corresponding Win32 function is TerminateProcess. In both cases, the killer must have the necessary authorization to do in the kill.

### *2.1.4 Process Hierarchies*

In some systems, when a process creates another process, the parent process and child process continue to be associated in certain ways. The child process can itself create more processes, forming a process hierarchy.

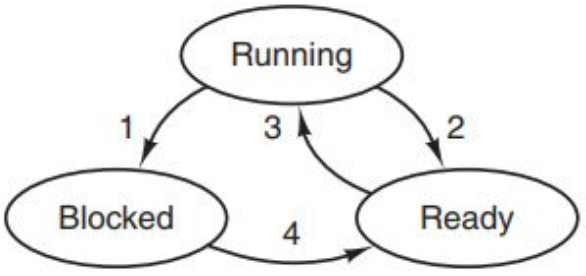
In UNIX, a process and all of its children and further descendants together form a process group. When a user sends a signal from the keyboard, the signal is delivered to all members of the process group currently associated with the keyboard
Individually, each process can catch the signal, ignore the signal, or take the default action, which is to be killed by the signal.

Windows has no concept of a process hierarchy.

### *2.1.5 Process States*

When a process blocks, it does so because logically it cannot continue, typically because it is waiting for input that is not yet available. It is also possible for a process that is conceptually ready and able to run to be stopped because the operating system has decided to allocate the CPU to another process for a while.

States of a process:
1. Running (actually using the CPU at that instant).
2. Ready (runnable; temporarily stopped to let another process run).
3. Blocked (unable to run until some external event happens).

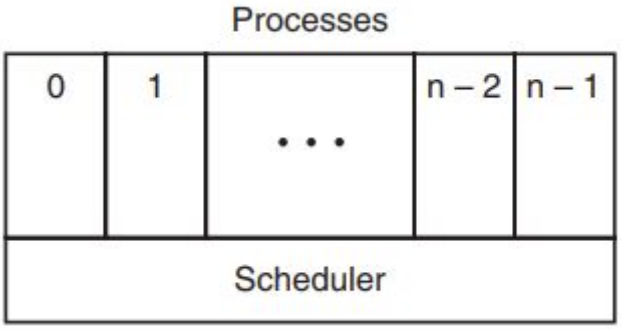



1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available

Transition 1 occurs when the operating system discovers that a process cannot continue right now. In some systems the process can execute a system call, such as pause, to get into blocked state. In other systems, including UNIX, when a process reads from a pipe or special file (e.g., a terminal) and there is no input available, the process is automatically blocked.

Transitions 2 and 3 are caused by the process scheduler, a part of the operating system, without the process even knowing about them. Transition 2 occurs when the scheduler decides that the running process has run long enough, and it is time to let another process have some CPU time. Transition 3 occurs when all the other processes have had their fair share and it is time for the first process to get the CPU to run again.

Transition 4 occurs when the external event for which a process was waiting (such as the arrival of some input) happens. If no other process is running at that instant, transition 3 will be triggered and the process will start running. Otherwise it may have to wait in ready state for a little while until the CPU is available and its turn comes.




Here the lowest level of the operating system is the scheduler, with a variety of processes on top of it. All the interrupt handling and details of actually starting and stopping processes are hidden away in what is here called the scheduler, which is actually not much code. The rest of the operating system is nicely structured in process form.

### *2.1.6 Implementation of Processes*

To implement the process model, the operating system maintains a table (an array of structures), called the **process table**, with one entry per process. (Some authors call these entries **process control blocks**.)

This entry contains important information about the process' state, including its program counter, stack pointer, memory allocation, the status of its open files, its accounting and scheduling information, and everything else about the process that must be saved when the process is switched from running to ready or blocked state so that it can be restarted later as if it had never been stopped.

Associated with each I/O class is a location called the interrupt vector. It contains the address of the interrupt service procedure.

Suppose that user process 3 is running when a disk interrupt happens. User process 3's program counter, program status word, and sometimes one or more registers are pushed onto the (current) stack by the interrupt hardware. The computer then jumps to the address specified in the interrupt vector. That is all the hardware does. From here on, it is up to the software, in particular, the interrupt service procedure.

All interrupts start by saving the registers, often in the process table entry for the current process. Then the information pushed onto the stack by the interrupt is removed and the stack pointer is set to point to a temporary stack used by the process handler.

When this routine is finished, it calls a C procedure to do the rest of the work for this specific interrupt type. When it has done its job, possibly making some process now ready, the scheduler is called to see who to run next. After that, control is passed back to the assembly-language code to load up the registers and memory map for the now-current process and start it running.

A process may be interrupted thousands of times during its execution, but the key idea is that after each interrupt the interrupted process returns to precisely the same state it was in before the interrupt occurred.

1. Hardware stacks program counter, etc.
2. Hardware loads new program counter from interrupt vector.
3. Assembly-language procedure saves registers.
4. Assembly-language procedure sets up new stack.
5. C interrupt service runs (typically reads and buffers input).
6. Scheduler decides which process is to run next.
7. C procedure returns to the assembly code.
8. Assembly-language procedure starts up new current process.

**Figure 2-5.** Skeleton of what the lowest level of the operating system does when an interrupt occurs.

### *2.2 THREADS*

In traditional operating systems, each process has an address space and a single thread of control. In fact, that is almost the definition of a process. Nevertheless, in many situations, it is desirable to have multiple threads of control in the same address space running in quasi-parallel, as though they were (almost) separate processes (except for the shared address space).

### *2.2.1 Thread Usage*

There are several reasons for having these mini processes, called **threads**.

- By decomposing such an application into multiple sequential threads that run in quasi-parallel, the programming model becomes simpler.
- A second argument for having threads is that since they are lighter weight than processes, they are easier (i.e., faster) to create and destroy than processes.
- Threads yield no performance gain when all of them are CPU bound, but when there is substantial computing and also substantial I/O, having threads allows these activities to overlap, thus speeding up the application.
- Threads are useful on systems with multiple CPUs, where real parallelism is possible.

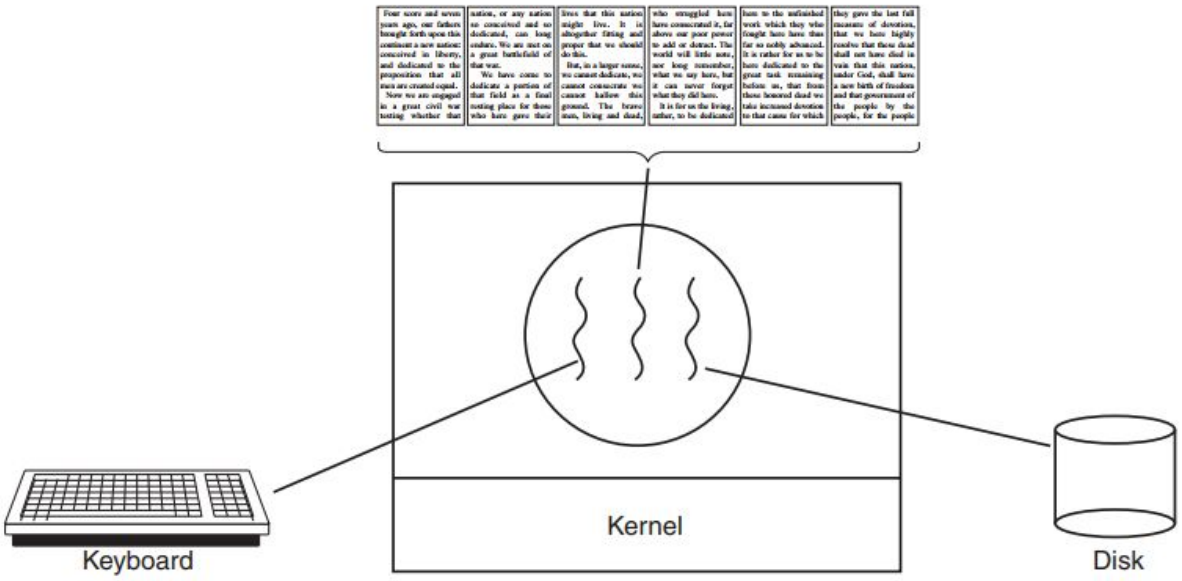
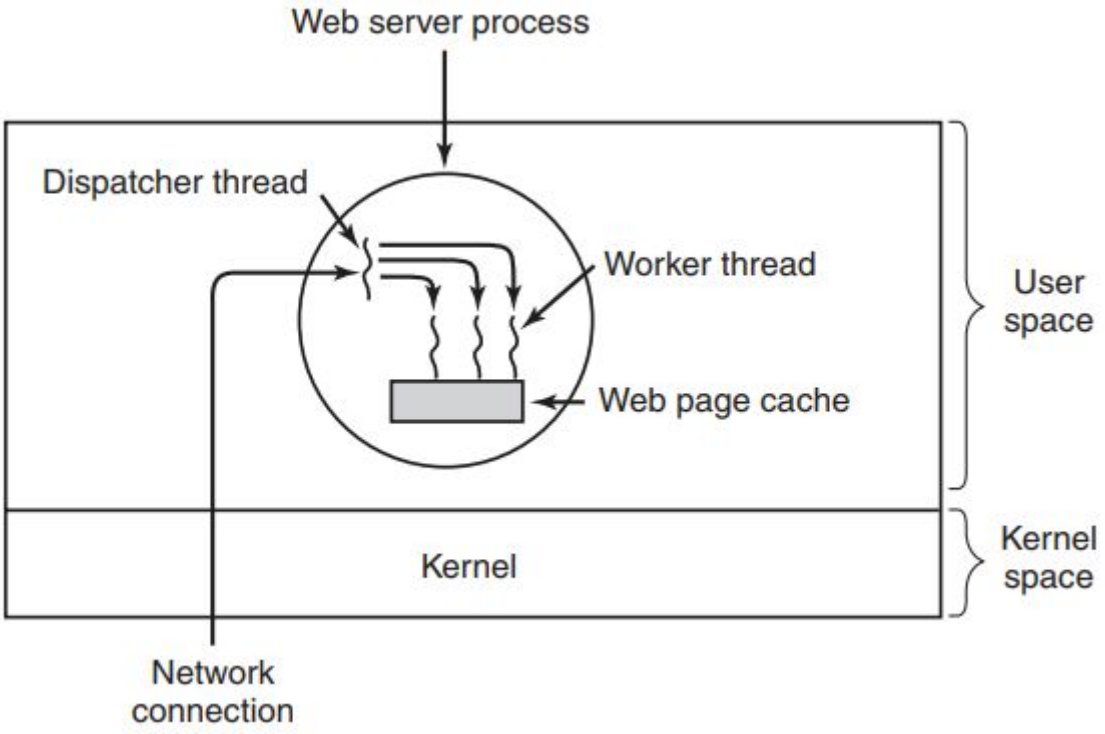



**Figure 2-7.** A word processor with three threads.

It should be clear that having three separate processes would not work here because all three threads need to operate on the document. By having three threads instead of three processes, they share a common memory and thus all have access to the document being edited. With three processes this would be impossible.

Web servers use this fact to improve performance by maintaining a collection of heavily used pages in main memory to eliminate the need to go to disk to get them. Such a collection is called a **cache** and is used in many other contexts as well.




Here one thread, the dispatcher, reads incoming requests for work from the network. After examining the request, it chooses an idle (i.e., blocked) worker thread and hands it the request, possibly by writing a pointer to the message into a special word associated with each thread. The dispatcher then wakes up the sleeping worker, moving it from blocked state to ready state.

When the worker wakes up, it checks to see if the request can be satisfied from the Web page cache, to which all threads have access. If not, it starts a read operation to get the page from the disk and blocks until the disk operation completes.

Suppose that threads are not available but the system designers find the performance loss due to single threading unacceptable. If a nonblocking version of the read system call is available, a third approach is possible. When a request comes in, the one and only thread examines it. If it can be satisfied from the cache, fine, but if not, a nonblocking disk operation is started.
The server records the state of the current request in a table and then goes and gets the next event. The next event may either be a request for new work or a reply from the disk about a previous operation. If it is new work, that work is started. If it is a reply from the disk, the relevant information is fetched from the table and the reply processed. With nonblocking disk I/O, a reply probably will have to take the form of a signal or interrupt. In this design, the ‘‘sequential process’’ model that we had in the first two cases is lost. The state of the computation must be explicitly saved and restored in the table every time the server switches from working on one request to another. In effect, we are simulating the threads and their stacks the hard way. A design like this, in which each computation has a saved state, and there exists some set of events that can occur to change the state, is called a f**inite-state machine.**

| Model | Characteristics |
|---|---|
| Threads | Parallelism, blocking system calls |
| Single-threaded process | No parallelism, blocking system calls |
| Finite-state machine | Parallelism, nonblocking system calls, interrupts |

It should now be clear what threads have to offer. They make it possible to retain the idea of sequential processes that make blocking calls (e.g., for disk I/O) and still achieve parallelism. Blocking system calls make programming easier, and parallelism improves performance. The single-threaded server retains the simplicity of blocking system calls but gives up performance. The third approach achieves high performance through parallelism but uses nonblocking calls and interrupts and thus is hard to program.

Having the CPU go idle when there is lots of computing to do is clearly wasteful and should be avoided if possible.

### *2.2.2 The Classical Thread Model*

One way of looking at a process is that it is a way to **group related resources** together. A process has an address space containing program text and data, as well as other resources. These resources may include open files, child processes, pending alarms, signal handlers, accounting information, and more. By putting them together in the form of a process, they can be managed more easily.

The other concept a process has is a thread of **execution**, usually shortened to just thread. The thread has a program counter that keeps track of which instruction to execute next. It has registers, which hold its current working variables. It has a stack, which contains the execution history, with one frame for each procedure called but not yet returned from. Although a thread must execute in some process, the thread and its process are different concepts and can be treated separately. Processes are used to group resources together; threads are the entities scheduled for execution on the CPU.

What threads add to the process model is to allow multiple executions to take place in the same process environment.

The term **multithreading** is also used to describe the situation of allowing multiple threads in the same process.

When a multithreaded process is run on a single-CPU system, the threads take turns running. By switching back and forth among multiple processes, the system gives the illusion of separate sequential processes running in parallel. Multithreading works the same way. The CPU switches rapidly back and forth among the threads, providing the illusion that the threads are running in parallel, albeit on a slower CPU than the real one.

With three compute-bound threads in a process, the threads would appear to be running in parallel, each one on a CPU with one-third the speed of the real CPU.

Different threads in a process are not as independent as different processes. All threads have exactly the same address space, which means that they also share the same global variables. Since every thread can access every memory address within the process' address space, one thread can read, write, or even wipe out another thread's stack. There is no protection between threads because (1) it is impossible, and (2) it should not be necessary.

Unlike different processes, which may be from different users and which may be hostile to one another, a process is always owned by a single user, who has presumably created multiple threads so that they can cooperate, not fight.

In addition to sharing an address space, all the threads can share the same set of open files, child processes, alarms, and signals, an so on.

| **Per-process items** | **Per-thread items** |
|---|---|
| Address space | Program counter |
| Global variables | Registers |
| Open files | Stack |
| Child processes | State |
| Pending alarms | |
| Signals and signal handlers | |
| Accounting information | |

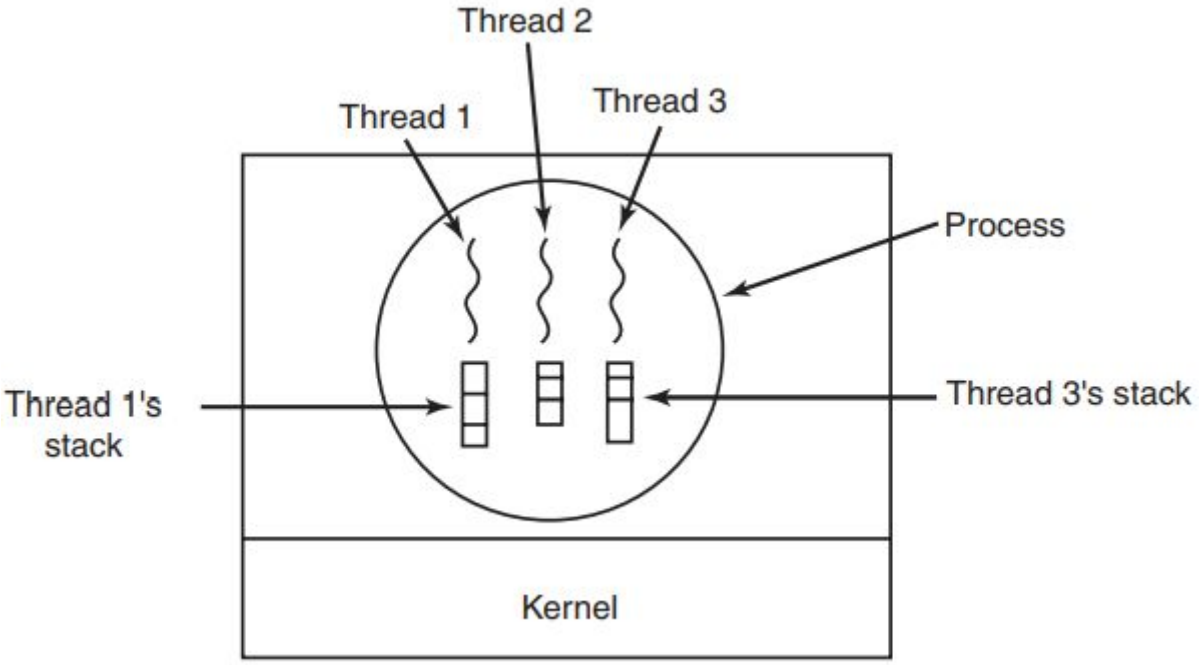
What we are trying to achieve with the thread concept is the ability for multiple threads of execution to share a set of resources so that they can work together closely to perform some task.

Like a traditional process (i.e., a process with only one thread), a thread can be in any one of several states: running, blocked, ready, or terminated. A running thread currently has the CPU and is active. In contrast, a blocked thread is waiting for some event to unblock it. A ready thread is scheduled to run and will as soon as its turn comes up.

It is important to realize that each thread has its own **stack**.

Each thread's stack contains one frame for each procedure called but not yet returned from. This frame contains the procedure's local variables and the return address to use when the procedure call has finished.

Each thread will generally call different procedures and thus have a different execution history. This is why each thread needs its own stack.




When multithreading is present, processes usually start with a single thread present. This thread has the ability to create new threads by calling a library procedure such as **thread_create**. A parameter to thread create specifies the name of a procedure for the new thread to run.

Sometimes threads are hierarchical, with a parent-child relationship, but often no such relationship exists, with all threads being equal. With or without a hierarchical relationship, the creating thread is usually returned a thread identifier that names the new thread.

When a thread has finished its work, it can exit by calling a library procedure, say, **thread_exit**. It then vanishes and is no longer schedulable.

In some thread systems, one thread can wait for a (specific) thread to exit by calling a procedure, for example, **thread_join**. This procedure blocks the calling thread until a (specific) thread has exited.

Another common thread call is **thread_yield**, which allows a thread to voluntarily give up the CPU to let another thread run. Such a call is important because there is no clock interrupt to actually enforce multiprogramming as there is with processes. Thus it is important for threads to be polite and voluntarily surrender the CPU from time to time to give other threads a chance to run.

### *2.2.3 POSIX Threads*

| Thread call | Description |
|---|---|
| Pthread_create | Create a new thread |
| Pthread_exit | Terminate the calling thread |
| Pthread_join | Wait for a specific thread to exit |
| Pthread_yield | Release the CPU to let another thread run |
| Pthread_attr_init | Create and initialize a thread's attribute structure |
| Pthread_attr_destroy | Remove a thread's attribute structure |

All Pthreads threads have certain properties. Each one has an identifier, a set of registers (including the program counter), and a set of attributes, which are stored in a structure. The attributes include the stack size, scheduling parameters, and other items needed to use the thread.

**Pthread_attr_init** creates the attribute structure associated with a thread and initializes it to the default values. These values (such as the priority) can be changed by manipulating fields in the attribute structure

**pthread_attr_destroy** removes a thread's attribute structure, freeing up its memory. It does not affect threads using it; they continue to exist.

2.2.4 Implementing Threads in User Space

There are two main places to implement threads: user space and the kernel.

The first method is to put the threads package entirely in user space. The kernel knows nothing about them. As far as the kernel is concerned, it is managing ordinary, single-threaded processes.

A user-level threads package can be implemented on an operating system that does not support threads

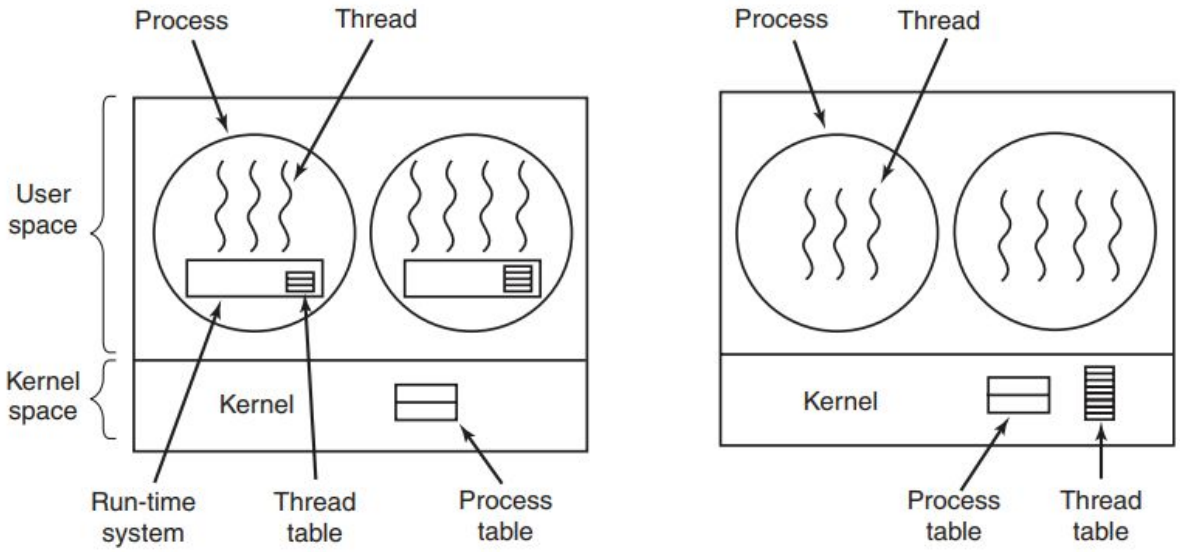



**Figure 2-16.** (a) A user-level threads package. (b) A threads package managed by the kernel.

When threads are managed in user space, each process needs its own private **thread table** to keep track of the threads in that process.

This table is analogous to the kernel's process table, except that it keeps track only of the per-thread properties, such as each thread's program counter, stack pointer, registers, state, and so forth.
When a thread is moved to ready state or blocked state, the information needed to restart it is stored in the thread table, exactly the same way as the kernel stores information about processes in the process table.

When a thread does something that may cause it to become blocked locally, for example, waiting for another thread in its process to complete some work, it calls a run-time system procedure. This procedure checks to see if the thread must be put into blocked state. If so, it stores the thread's registers (i.e., its own) in the thread table, looks in the table for a ready thread to run, and reloads the machine registers with the new thread's saved values. As soon as the stack pointer and program counter have been switched, the new thread comes to life again automatically.

The key difference with processes:  When a thread is finished running for the moment, for example, when it calls thread yield, the code of thread yield can save the thread's information in the thread table itself. Furthermore, it can then call the thread scheduler to pick another thread to run.

User-level threads also have other advantages. They allow each process to have its own customized scheduling algorithm.

Despite their better performance, user-level threads packages have some major problems. First among these is the problem of how blocking system calls are implemented.
Letting the thread actually make the system call is unacceptable, since this will stop all the threads.

The system calls could all be changed to be nonblocking, but requiring changes to the operating system is unattractive.

Another alternative is available in the event that it is possible to tell in advance if a call will block. In most versions of UNIX, a system call, **select**, exists, which allows the caller to tell whether a prospective read will block. When this call is present, the library procedure read can be replaced with a new one that first does a select call and then does the read call only if it is safe (i.e., will not block).

The code placed around the system call to do the checking is called a **jacket** or **wrapper**.

**Page fault**: Computers can be set up in such a way that not all of the program is in main memory at once. If the program calls or jumps to an instruction that is not in memory, a page fault occurs and the operating system will go and get the missing instruction (and its neighbors) from disk.

The process is blocked while the necessary instruction is being located and read in. If a thread causes a page fault, the kernel, unaware of even the existence of threads, naturally blocks the entire process until the disk I/O is complete, even though other threads might be runnable.

Another problem with user-level thread packages is that if a thread starts running, no other thread in that process will ever run unless the first thread voluntarily gives up the CPU.

Another, and really the most devastating, argument against user-level threads is that programmers generally want threads precisely in applications where the threads block often, as, for example, in a multithreaded Web server.

Advantages:

- A user-level threads package can be implemented on an operating system that does not support threads
- They allow each process to have its own customized scheduling algorithm.

Disadvantages:

- the problem of how blocking system calls are implemented.
  Letting the thread actually make the system call is unacceptable, since this will stop all the threads.
- If a thread causes a page fault, the kernel, unaware of even the existence of threads, naturally blocks the entire process until the disk I/O is complete, even though other threads might be runnable.
- if a thread starts running, no other thread in that process will ever run unless the first thread voluntarily gives up the CPU.
- programmers generally want threads precisely in applications where the threads block often, as, for example, in a multithreaded Web server.

### *2.2.5 Implementing Threads in the Kernel*

Now let us consider having the kernel know about and manage the threads.

No run-time system is needed.
There is no thread table in each process.
The kernel has a thread table that keeps track of all the threads in the system.

When a thread wants to create a new thread or destroy an existing thread, it makes a kernel call, which then does the creation or destruction by updating the kernel thread table.

The kernel's thread table holds each thread's registers, state, and other information. The information is the same as with user-level threads, but now kept in the kernel instead of in user space.

This information is a subset of the information that traditional kernels maintain about their single-threaded processes, that is, the process state. In addition, the kernel also maintains the traditional process table to keep track of processes.

All calls that might block a thread are implemented as system calls.

When a thread blocks, the kernel, at its option, can run either another thread from the same process (if one is ready) or a thread from a different process. With user-level threads, the run-time system keeps running threads from its own process until the kernel takes the CPU away from it (or there are no ready threads left to run).

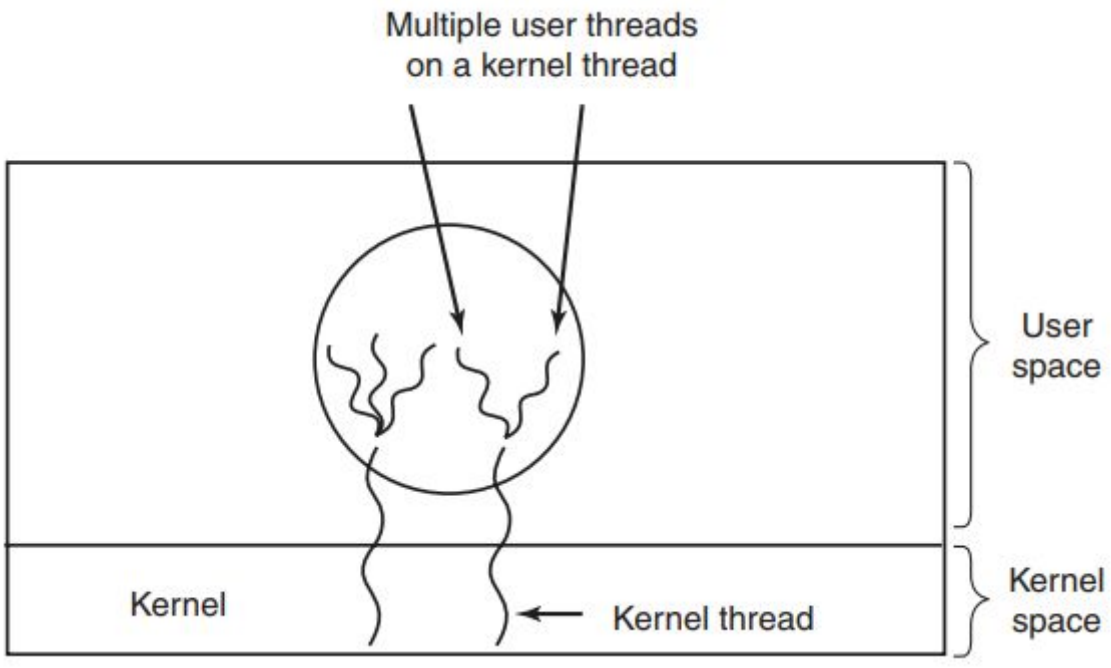
Due to the relatively greater cost of creating and destroying threads in the kernel, some systems take an environmentally correct approach and recycle their threads. When a thread is destroyed, it is marked as not runnable, but its kernel data structures are not otherwise affected. Later, when a new thread must be created, an old thread is reactivated, saving some overhead. Thread recycling is also possible for user-level threads, but since the thread-management overhead is much smaller, there is less incentive to do this.

Kernel threads do not require any new, nonblocking system calls. In addition, if one thread in a process causes a page fault, the kernel can easily check to see if the process has any other runnable threads, and if so, run one of them while waiting for the required page to be brought in from the disk. Their main disadvantage is that the cost of a system call is substantial, so if thread operations (creation, termination, etc.) a common, much more overhead will be incurred.

2.2.6 Hybrid Implementations

Various ways have been investigated to try to combine the advantages of user level threads with kernel-level threads. One way is use kernel-level threads and then multiplex user-level threads onto some or all of them.

When this approach is used, the programmer can determine how many kernel threads to use and how many user-level threads to multiplex on each one. This model gives the ultimate in flexibility.




With this approach, the kernel is aware of only the kernel-level threads and schedules those. Some of those threads may have multiple user-level threads multiplexed on top of them. These user-level threads are created, destroyed, and scheduled just like user-level threads

in a process that runs on an operating system without multithreading capability. In this model, each kernel-level thread has some set of user-level threads that take turns using it.

### 2.2.7 Scheduler Activations

While kernel threads are better than user-level threads in some key ways, they are also indisputably slower.

The goals of the **scheduler activation** work are to mimic the functionality of kernel threads, but with the better performance and greater flexibility usually associated with threads packages implemented in user space. In particular, user threads should not have to make special nonblocking system calls or check in advance if it is safe to make certain system calls. Nevertheless, when a thread blocks on a system call or on a page fault, it should be possible to run other threads within the same process, if any are ready.

Efficiency is achieved by avoiding unnecessary transitions between user and kernel space. If a thread blocks waiting for another thread to do something, for example, there is no reason to involve the kernel, thus saving the overhead of the kernel-user transition.

When scheduler activations are used, the kernel assigns a certain number of virtual processors to each process and lets the (user-space) run-time system allocate threads to processors.

The basic idea that makes this scheme work is that when the kernel knows that a thread has blocked, the kernel notifies the process' run-time system, passing as parameters on the stack the number of the thread in question and a description of the event that occurred. The notification happens by having the kernel activate the run-time system at a known starting address. This mechanism is called an **upcall**.

Once activated, the run-time system can reschedule its threads, typically by marking the current thread as blocked and taking another thread from the ready list, setting up its registers, and restarting it. Later, when the kernel learns that the original thread can run again (e.g., the pipe it was trying to read from now contains data, or the page it faulted over has been brought in from disk), it makes another upcall to the run-time system to inform it. The run-time system can either restart the blocked thread immediately or put it on the ready list to be run later.

When a hardware interrupt occurs while a user thread is running, the interrupted CPU switches into kernel mode. If the interrupt is caused by an event not of interest to the interrupted process, such as completion of another process' I/O, when the interrupt handler has finished, it puts the interrupted thread back in the state it was in before the interrupt. If, however, the process is interested in the interrupt, such as the arrival of a page needed by one of the process' threads, the interrupted thread is not restarted. Instead, it is suspended, and the run-time system is started on that virtual CPU, with the state of the interrupted thread on the stack. It is then up to the run-time system to decide which thread to schedule on that CPU: the interrupted one, the newly ready one, or some third choice.

### *2.2.8 Pop-Up Threads*

An important example is how incoming messages, for example requests for service, are handled. The traditional approach is to have a process or thread that is blocked on a receive system call waiting for an incoming message. When a message arrives, it accepts the message, unpacks it, examines the contents, and processes it.

However, a completely different approach is also possible, in which the arrival of a message causes the system to create a new thread to handle the message. Such a thread is called a pop-up thread.

A key advantage of pop-up threads is that since they are brand new, they do not have any history—registers, stack, whatever—that must be restored. Each one starts out fresh and each one is identical to all the others. This makes it possible to create such a thread quickly. The new thread is given the incoming message to process. The result of using pop-up threads is that the latency between message arrival and the start of processing can be made very short.

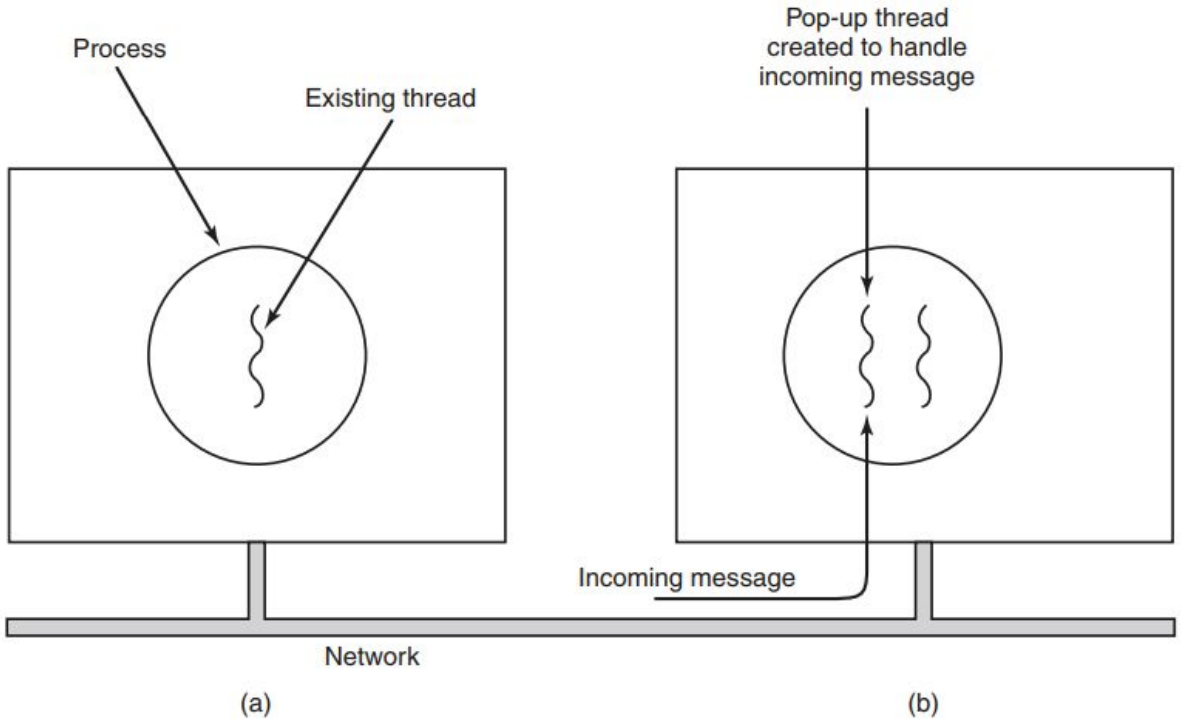



**Figure 2-18.** Creation of a new thread when a message arrives. (a) Before the message arrives. (b) After the message arrives.

If the system supports threads running in the kernel's context, the thread may run there. Having the pop-up thread run in kernel space is usually easier and faster than putting it in user space. A pop-up thread in kernel space can easily access all the kernel's tables and the I/O devices, which may be needed for interrupt processing.

A buggy kernel thread can do more damage than a buggy user thread. For example, if it runs too long and there is no way to preempt it, incoming data may be permanently lost.

### *2.2.9 Making Single-Threaded Code Multithreaded*

Many existing programs were written for single-threaded processes. Converting these to multithreading is much trickier than it may at first appear.

As a start, the code of a thread normally consists of multiple procedures, just like a process. These may have local variables, global variables, and parameters. Local variables and parameters do not cause any trouble, but variables that are global to a thread but not global to the entire program are a problem. These are variables that are global in the sense that many procedures within the thread use them (as they might use any global variable), but other threads should logically leave them alone.

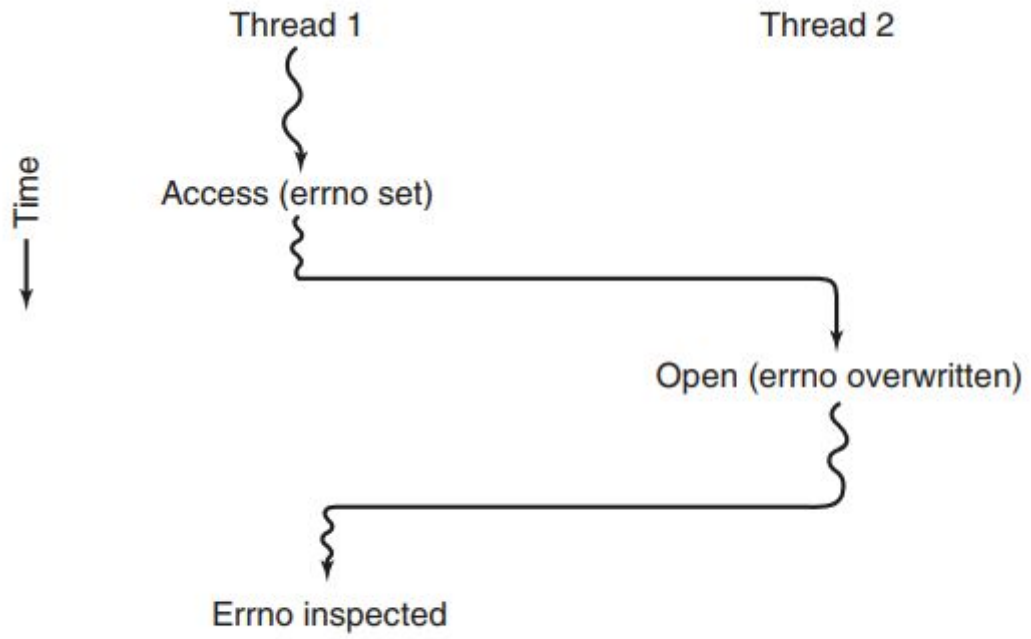
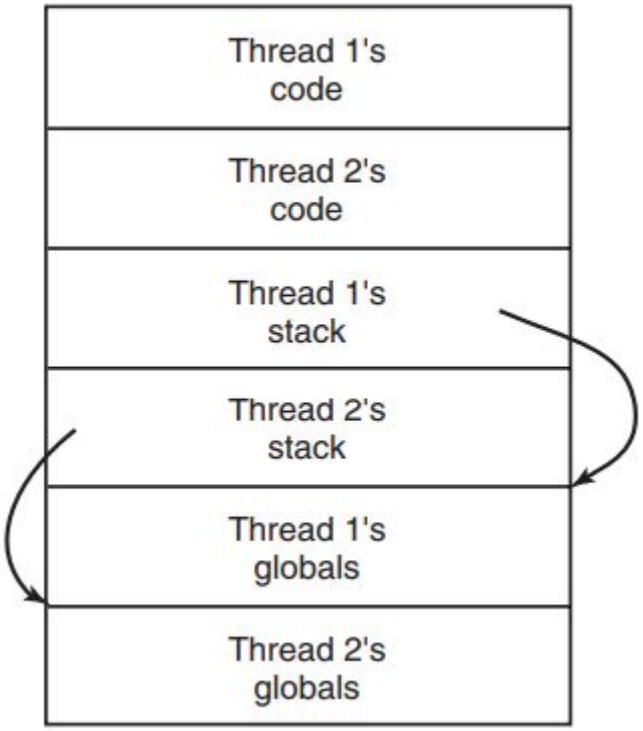



**Figure 2-19.** Conflicts between threads over the use of a global variable.

A solution is to assign each thread its own private global variables. In this way, each thread has its own private copy of errno and other global variables, so conflicts are avoided.




Accessing the private global variables is a bit tricky, however, since most programming languages have a way of expressing local variables and global variables, but not intermediate forms.

new library procedures can be introduced to create, set, and read these threadwide global variables. The first call might look like this:

**create_global("bufptr");**

It allocates storage for a pointer called bufptr on the heap or in a special storage area reserved for the calling thread. No matter where the storage is allocated, only the calling thread has access to the global variable. If another thread creates a global variable with the same name, it gets a different storage location that does not conflict with the existing one.

Two calls are needed to access global variables: one for writing them and the other for reading them. For writing, something like:

**set global("bufptr", &buf);**

will do. It stores the value of a pointer in the storage location previously created by the call to create global. To read a global variable, the call might look like

**bufptr = read global("bufptr");**

It returns the address stored in the global variable, so its data can be accessed.

The next problem in turning a single-threaded program into a multithreaded one is that many library procedures are not reentrant. That is, they were not designed to have a second call made to any given procedure while a previous call has not yet finished.

While malloc is busy updating these lists, they may temporarily be in an inconsistent state, with pointers that point nowhere. If a thread switch occurs while the tables are inconsistent and a new call comes in from a different thread, an invalid pointer may be used, leading to a program crash. Fixing all these problems effectively means rewriting the entire library.

A different solution is to provide each procedure with a jacket that sets a bit to mark the library as in use. Any attempt for another thread to use a library procedure while a previous call has not yet completed is blocked. Although this approach can be made to work, it greatly eliminates potential parallelism.

Problems:

- Another thread writing to a global variable that tread 1 wants to read resulting in wrong data.

- Many library procedures are not reentrant. That is, they were not designed to have a second call made to any given procedure while a previous call has not yet finished.

- If a thread switch occurs while the tables are inconsistent and a new call comes in from a different thread, an invalid pointer may be used, leading to a program crash.

- Some signals are logically thread specific, whereas others are not.

- In many systems, when a process' stack overflows, the kernel just provides that process with more stack automatically. When a process has multiple threads, it must also have multiple stacks. If the kernel is not aware of all these stacks, it cannot grow them automatically upon stack fault. In fact, it may not even realize that a memory fault is related to the growth of some thread's stack.

### *2.3 INTERPROCESS COMMUNICATION*

three issues with interprocess communication:

- how one process can pass information to another
- making sure two or more processes do not get in each other's way
- proper sequencing when dependencies are present

#### *2.3.1 Race Conditions*

How does interprocess communication work:
When a process wants to print a file, it enters the file name in a special **spooler directory**. Another process, the **printer daemon**, periodically checks to see if there are any files to be printed, and if there are, it prints them and then removes their names from the directory.

Process A reads in = 2, gets interrupted by CPU.
Process B reads in = 2, puts file name there, gets interrupted.
Process A starts where it left off, and puts file name in 2, overwriting the file name of process B
This is called a **race condition**. The file of process B will never get printed.

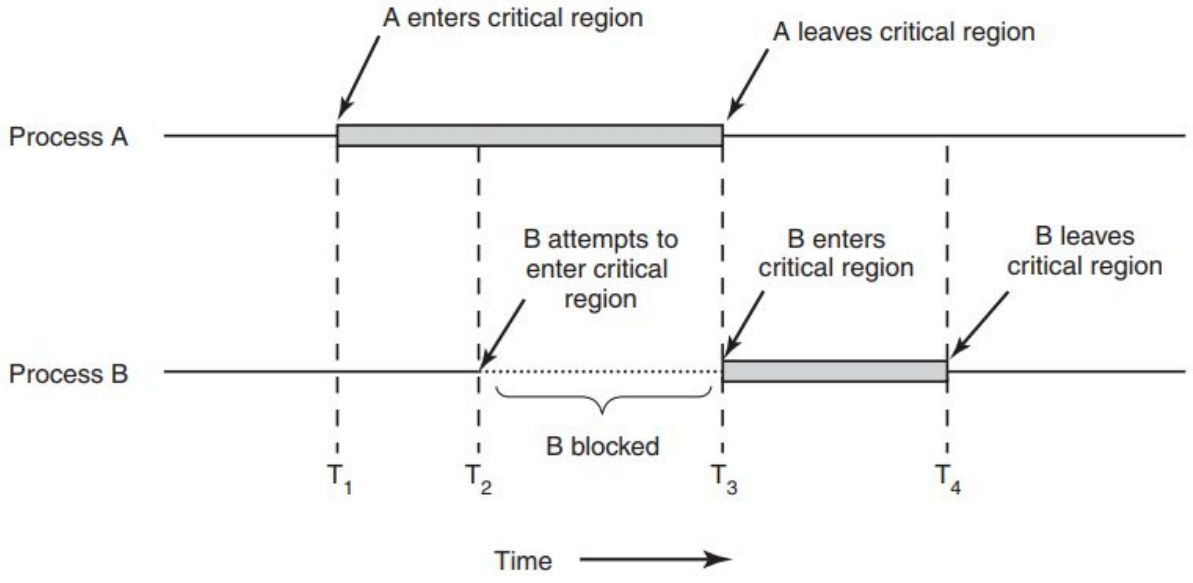
2.3.2 Critical Regions

How to avoid race conditions.
We need **mutual exclusion**, some way of making sure that if one process is using a shared variable or file, the other processes will be excluded from doing the same thing.

The part of the program where the shared memory is accessed is called the **critical region** or **critical section**.

Four conditions to avoid race conditions:

- No two processes may be simultaneously inside their critical regions
- No assumptions may be made about speeds or the number of CPUs.
- No process running outside its critical region may block any process.
- No process should have to wait forever to enter its critical region.




### *2.3.3 Mutual Exclusion with Busy Waiting*

**Disabling Interrupts**

On a single-processor system, the simplest solution is to have each process disable all interrupts just after entering its critical region and re-enable them just before leaving it. With interrupts disabled, no clock interrupts can occur. The CPU is only switched from process to process as a result of clock or other interrupts.

This approach is generally unattractive because it is unwise to give user processes the power to turn off interrupts. What if one of them did it, and never turned them on again? That could be the end of the system.

if the system is a multiprocessor (with two or more CPUs) disabling interrupts affects only the CPU that executed the disable instruction. The other ones will continue running and can access the shared memory.

**Lock Variables**

Consider having a single, shared (lock) variable, initially 0. When a process wants to enter its critical region, it first tests the lock. If the lock is 0, the process sets it to 1 and enters the critical region. If the lock is already 1, the process just waits until it becomes 0. Thus, a 0 means that no process is in its critical region, and a 1 means that some process is in its critical region.

this idea contains exactly the same fatal flaw that we saw in the spooler directory. Suppose that one process reads the lock and sees that it is 0. Before it can set the lock to 1, another process is scheduled, runs, and sets the lock to 1. When the first process runs again, it will also set the lock to 1, and two processes will be in their critical regions at the same time.

**Strict Alternation**

the integer variable turn, initially 0, keeps track of whose turn it is to enter the critical region and examine or update the shared memory. Initially, process 0 inspects turn, finds it to be 0, and enters its critical region. Process 1 also finds it to be 0 and therefore sits in a tight loop continually testing turn to see when it becomes 1. Continuously testing a variable until some value appears is called **busy waiting**. It should usually be avoided, since it wastes CPU time. Only when there is a reasonable expectation that the wait will be short is busy waiting used. A lock that uses busy waiting is called a **spin lock**.

**Peterson's Solution**

Before using the shared variables (i.e., before entering its critical region), each process calls enter region with its own process number, 0 or 1, as parameter. This call will cause it to wait, if need be, until it is safe to enter. After it has finished with the shared variables, the process calls leave region to indicate that it is done and to allow the other process to enter, if it so desires.

**The TSL Instruction**

*TSL RX,LOCK*

(Test and Set Lock) that works as follows. It reads the contents of the memory word lock into register RX and then stores a nonzero value at the memory address lock.The CPU executing the TSL instruction locks the memory bus to prohibit other CPUs from accessing memory until it is done.

Disabling interrupts on processor 1 has no effect at all on processor 2. The only way to keep processor 2 out of the memory until processor 1 is finished is to lock the bus, which requires a special hardware facility (basically, a bus line asserting that the bus is locked and not available to processors other than the one that locked it).

To use the TSL instruction, we will use a shared variable, lock, to coordinate access to shared memory. When lock is 0, any process may set it to 1 using the TSL instruction and then read or write the shared memory. When it is done, the process sets lock back to 0 using an ordinary move instruction.

2.3.4 Sleep and Wakeup

The scheduling rules are such that H is run whenever it is in ready state. At a certain moment, with L in its critical region, H becomes ready to run (e.g., an I/O operation completes). H now begins busy waiting, but since L is never scheduled while H is running, L never gets the chance to leave its critical region, so H loops forever. This situation is sometimes referred to as the priority **inversion problem**.

Sleep is a system call that causes the caller to block, that is, be suspended until another process wakes it up.

**The Producer-Consumer Problem**

As an example of how these primitives can be used, let us consider the producer-consumer problem (also known as the bounded-buffer problem). Two processes share a common, fixed-size buffer. One of them, the producer, puts information into the buffer, and the other one, the consumer, takes it out.

The problem is that when the consumer reads 0, then the CPU blocks it and the producer puts an item in the buffer. The producer will send wake up call to the consumer. Since the consumer wasn't sleeping yet this wake up call is lost. The consumer gets the CPU and goes to sleep since it has read a 0.

A quick fix is to modify the rules to add a **wakeup waiting bit** to the picture. When a wakeup is sent to a process that is still awake, this bit is set. Later, when the process tries to go to sleep, if the wakeup waiting bit is on, it will be turned off, but the process will stay awake.

with multiple processes a wakeup waiting bit is insufficient.

2.3.5 Semaphores

In his proposal, a new variable type, which he called a semaphore, was introduced. A **semaphore** could have the value 0, indicating that no wakeups were saved, or some positive value if one or more wakeups were pending.

Checking the value, changing it, and possibly going to sleep, are all done as a single, indivisible **atomic action**.

if the producer makes an up operation, one of the consumers that were sleeping on this semaphore is chosen at random to perform its down operation.

Semaphores that are initialized to 1 and used by two or more processes to ensure that only one of them can enter its critical region at the same time are called **binary semaphores**.

in a producent consumer problem we have thre
e semaphores:  one called **full** for counting the number of slots that are full, one called **empty** for counting the number of slots that are empty, and one called **mutex** to make sure the producer and consumer do not access the buffer at the same time.

A mutex is a shared variable that can be in one of two states: unlocked or locked. Consequently, only 1 bit is required to represent it, but in practice an integer often is used, with 0 meaning unlocked and all other values meaning locked.

When a thread (or process) needs access to a critical region, it calls mutex lock. If the mutex is currently unlocked (meaning that the critical region is available), the call succeeds and the calling thread is free to enter the critical region.

On the other hand, if the mutex is already locked, the calling thread is blocked until the thread in the critical region is finished and calls mutex unlock. If multiple threads are blocked on the mutex, one of them is chosen at random and allowed to acquire the lock.

Because mutexes are so simple, they can easily be implemented in user space provided that a TSL or XCHG instruction is available.

difference between mutex_lock and enter_region:
If enter_region fails, it keeps spinning until the lock becomes free and it can go in critical region. If mutex_lock fails it calls a thread_yield, which gives another thread the opportunity to run on the CPU.

**Futex**
A futex is a feature of Linux that implements basic locking (much like a mutex) but avoids dropping into the kernel unless it really has to.

If the lock is not free the process gets put on a wait queue, instead of spin locking. If the lock becomes free again the process in the wait queue gets the lock. Putting in the waiting queue involves the kernel which is slow, but you had to spin lock anyway. If there is no contention, the kernel is not involved at all.

**Mutexes in Pthreads**
Pthreads provides a number of functions that can be used to synchronize threads.

There is also an option for trying to lock a mutex and failing with an error code instead of blocking if it is already blocked. This call is pthread mutex trylock. This call allows a thread to effectively do busy waiting if that is ever needed.

| Thread call | Description |
|---|---|
| Pthread_mutex_init | Create a mutex |
| Pthread_mutex_destroy | Destroy an existing mutex |
| Pthread_mutex_lock | Acquire a lock or block |
| Pthread_mutex_trylock | Acquire a lock or fail |
| Pthread_mutex_unlock | Release a lock |

Pthreads offers a second synchronization mechanism: condition variables. Mutexes are good for allowing or blocking access to a critical region. Condition variables allow threads to block due to some condition not being met.

As a simple example, consider the producer-consumer scenario again: one thread puts things in a buffer and another one takes them out. If the producer discovers that there are no more free slots available in the buffer, it has to block until one becomes available. Mutexes make it possible to do the check atomically without interference from other threads, but having discovered that the buffer is full, the producer needs a way to block and be awakened later. This is what condition variables allow.

Condition variables and mutexes are always used together. The pattern is for one thread to lock a mutex, then wait on a conditional variable when it cannot get what it needs. Eventually another thread will signal it and it can continue. The pthread cond wait call atomically unlocks the mutex it is holding. For this reason, the mutex is one of the parameters.

| Thread call | Description |
|---|---|
| Pthread_cond_init | Create a condition variable |
| Pthread_cond_destroy | Destroy a condition variable |
| Pthread_cond_wait | Block waiting for a signal |
| Pthread_cond_signal | Signal another thread and wake it up |
| Pthread_cond_broadcast | Signal multiple threads and wake all of them |

### *2.3.7 Monitors*

if the mutex gets incremented before empty you get a deadlock. both the consumer and producer block and there is no way to de-block them. So the order of downs is very important which makes it very hard to program.

A **monitor** is a collection of procedures, variables, and data structures that are all grouped together in a special kind of module or package.

Processes may call the procedures in a monitor whenever they want to, but they cannot directly access the monitor's internal data structures from procedures declared outside the monitor.

Monitors have an important property that makes them useful for achieving mutual exclusion: only one process can be active in a monitor at any instant.

It is up to the compiler to implement mutual exclusion on monitor entries, but a common way is to use a mutex or a binary semaphore.

How should the producer block when it finds the buffer full? The solution lies in the introduction of condition variables, along with two operations on them, wait and signal. When a monitor procedure discovers that it cannot continue (e.g., the producer finds the buffer full), it does a wait on some condition variable, say, full. This action causes the calling process to block. It also allows another process that had been previously prohibited from entering the monitor to enter now.

This other process, for example, the consumer, can wake up it's sleeping partner by doing a signal on the condition variable that its partner is waiting on. To avoid having two active processes in the monitor at the same time, we need a rule telling what happens after a signal.

If a signal is done on a condition variable on which several processes are waiting, only one of them, determined by the system scheduler, is revived.

if a condition variable is signaled with no one waiting on it, the signal is lost forever. In other words, the wait must come before the signal.

### *2.3.8 Message Passing*

That something else is **message passing**. This method of interprocess communication uses two primitives, send and receive, which, like semaphores and unlike monitors, are system calls rather than language constructs. As such, they can easily be put into library procedures, such as send(destination, &message); and receive(source, &message);

**Design Issues for Message-Passing Systems**

Messages can be lost in the network. To guard against lost messages, the sender and receiver can agree that as soon as a message has been received, the receiver will send back a special **acknowledgement message**

If this acknowledgement message is lost, the sender will send the same message again. We can give each message a unique code. If the code from the first message is identical to the second message, we simply ignore it.

Message systems also have to deal with the question of how processes are named, so that the process specified in a send or receive call is unambiguous. **Authentication** is also an issue in message systems: how can the client tell that it is communicating with the real file server, and not with an imposter?

A **mailbox** is a place to buffer a certain number of messages, typically specified when the mailbox is created. When mailboxes are used, the address parameters in the send and receive calls are mailboxes, not processes. When a process tries to send to a mailbox that is full, it is suspended until a message is removed from that mailbox, making room for a new one.

### *2.4 SCHEDULING*

When a computer is multiprogrammed, it frequently has multiple processes or threads competing for the CPU at the same time. This situation occurs whenever two or more of them are simultaneously in the ready state. If only one CPU is available, a choice has to be made which process to run next. The part of the operating system that makes the choice is called the **scheduler**, and the algorithm it uses is called the **scheduling algorithm.**

#### 2.4.1 Introduction to Scheduling

In addition to picking the right process to run, the scheduler also has to worry about making efficient use of the CPU because process switching is expensive. To start with, a switch from user mode to kernel mode must occur. Then the state of the current process must be saved, including storing its registers in the process table so they can be reloaded later. In some systems, the memory map (e.g., memory reference bits in the page table) must be saved as well. Next a new process must be selected by running the scheduling algorithm. After that, the memory management unit (MMU) must be reloaded with the memory map of the new process. Finally, the new process must be started. In addition to all that, the process switch may invalidate the memory cache and related tables, forcing it to be dynamically reloaded from the main memory twice (upon entering the kernel and upon leaving it). All in all, doing too many process switches per second can chew up a substantial amount of CPU time, so caution is advised.

**CPU-Bound processes**: spend most time computing
**I/O bound processes**: Spend most time waiting for I/O

When to make scheduling decisions:

- When a process is started it needs to decide wether to run parent or child
- When a process exits it needs to decide the next process
- When a process blocks on I/O, a semaphore or some other reason another process needs to run
- When an I/O interrupt occurs.

A **nonpreemptive scheduling** algorithm picks a process to run and then just lets it run until it blocks (either on I/O or waiting for another process) or voluntarily releases the CPU.

no scheduling decisions are made during clock interrupts. After clock-interrupt processing has been finished, the process that was running before the interrupt is resumed, unless a higher-priority process was waiting for a now-satisfied timeout.

a **preemptive scheduling** algorithm picks a process and lets it run for a maximum of some fixed time. If it is still running at the end of the time interval, it is suspended and the scheduler picks another process to run (if one is available). Doing preemptive scheduling requires having a clock interrupt occur at the end of the time interval to give control of the CPU back to the scheduler. If no clock is available, nonpreemptive scheduling is the only option.

Categories of Scheduling Algorithms

1. Batch.
2. Interactive.
3. Real time.

In batch systems, there are no users impatiently waiting at their terminals for a quick response to a short request. Consequently, nonpreemptive algorithms, or preemptive algorithms with long time periods for each process, are often acceptable. This approach reduces process switches and thus improves performance.

In an environment with interactive users, preemption is essential to keep one process from hogging the CPU and denying service to the others. Even if no process intentionally ran forever, one process might shut out all the others indefinitely due to a program bug. Preemption is needed to prevent this behavior. Servers also fall into this category, since they normally serve multiple (remote) users, all of whom are in a big hurry. Computer users are always in a big hurry.

In systems with real-time constraints, preemption is, oddly enough, sometimes not needed because the processes know that they may not run for long periods of time and usually do their work and block quickly. The difference with interactive systems is that real-time systems run only programs that are intended to further the application at hand. Interactive systems are general purpose and may run arbitrary programs that are not cooperative and even possibly malicious.

Scheduling Algorithm Goals

**All systems**

Fairness - giving each process a fair share of the CPU
Policy enforcement - seeing that stated policy is carried out
Balance - keeping all parts of the system busy

**Batch systems**

Throughput - maximize jobs per hour
Turnaround time - minimize time between submission and termination
CPU utilization - keep the CPU busy all the time

**Interactive systems**

Response time - respond to requests quickly
Proportionality - meet users' expectations

**Real-time systems**

Meeting deadlines - avoid losing data
Predictability - avoid quality degradation in multimedia systems

### *2.4.2 Scheduling in Batch Systems*

**First-Come, First-Served**

A single queue of processes exists. new jobs are added to the end of the queue. The first on the list in the queue is the next job that is ran by the CPU.

disadvantage: Compute bound processes slow down the I/O block processes a lot.

**Shortest Job First**

When compute times are known, we can run the shortest jobs first.

The mean turnaround time is (4a + 3b + 2c + d)/4
As you can see a contributes more to the average turnaround time, which is why a has to be the smallest.

Shortest job first is only optimal if all jobs are available simultaneously.

**Shortest remaining next time**

The algorithm always chooses the process with the shortest remaining run time.

### *2.4.3 Scheduling in Interactive Systems*

**Round-Robin Scheduling**
Each process is assigned a time interval, called its quantum, during which it is allowed to run. If the process is still running at the end of the quantum, the CPU is preempted and given to another process. If the process has blocked or finished before the quantum has elapsed, the CPU switching is done when the process blocks, of course.

All the scheduler needs to do is maintain a list of runnable processes.
When the process uses up its quantum, it is put on the end of the list.

if the quantum is at 4msec and the process switching takes 1msec, 20% of the CPU time is wasted switching processes which isn't very efficient. We can make the quantum longer but this means that all requests during the quantum are ignored until the quantum is reached. Small requests may be done last, which prefers a smaller quantum so they get done earlier.

Another factor is that if the quantum is set longer than the mean CPU burst, preemption will not happen very often. Instead, most processes will perform a blocking operation before the quantum runs out, causing a process switch. Eliminating preemption improves performance because process switches then happen only when they are logically necessary, that is, when a process blocks and cannot continue.

**Priority Scheduling**
Each process is assigned a priority, and the runnable process with the highest priority is allowed to run.

To avoid a process to run indefinitely the scheduler can reduce the priority of the running process with each clock tick.

Giving a process with small cpu bursts, like I/O processes, a higher priority is smart because they don't slow down other processes to much.

You can have priority groups, wherein processes have the same priority in a group and have round robin scheduler in each group.

**Multiple Queues**
Priority classes are made.
High priority classes get a large quanta to run longer.
Low priority classes get small quanta to run shorter.

**Shortest Process Next**
Interactive processes generally follow the pattern of wait for command, execute command, wait for command, execute command, etc. If we regard the execution of each command as a separate ‘‘job,’’ then we can minimize overall response time by running the shortest one first. The problem is figuring out which of the currently runnable processes is the shortest one.

**Guaranteed scheduling**
Works on promises. If n people use CPU everyone gets 1/n of the CPU power.
If n processes are running, each process should get 1/n of the CPU Cycles.

The algorithm is then to run the process with the lowest ratio until its ratio has moved above that of its closest competitor. Then that one is chosen to run next.

**lottery scheduling**
Give processes a lottery ticket for cpu time.
When applied to CPU scheduling, the system might hold a lottery 50 times a second, with each winner getting 20 msec of CPU time as a prize.

Each winner getting 20 msec of CPU time as a prize. To paraphrase George Orwell: ‘‘All processes are equal, but some processes are more equal.’’ More important processes can be given extra tickets, to increase their odds of winning. If there are 100 tickets outstanding, and one process holds 20 of them, it will have a 20% chance of winning each lottery. In the long run, it will get about 20% of the CPU.

Cooperating processes may exchange tickets if they wish. For example, when a client process sends a message to a server process and then blocks, it may give all of its tickets to the server, to increase the chance of the server running next. When the server is finished, it returns the tickets so that the client can run again. In fact, in the absence of clients, servers need no tickets at all.

**Fair-Share Scheduling**
if two users have each been promised 50% of the CPU, they will each get that, no matter how many processes they have in existence.

### *2.4.4 Scheduling in Real-Time Systems*
A real-time system is one in which time plays an essential role. Typically, one or more physical devices external to the computer generate stimuli, and the computer must react appropriately to them within a fixed amount of time.

the computer in a compact disc player gets the bits as they come off the drive and must convert them into music within a very tight time interval. If the calculation takes too long, the music will sound peculiar.

Real-time systems are generally categorized as **hard real time**, meaning there are absolute deadlines that must be met—or else!— and **soft real time**, meaning that missing an occasional deadline is undesirable, but nevertheless tolerable.

The scheduler has to schedule processes in a way that all process meet their deadline

The events that a real-time system may have to respond to can be further categorized as periodic (meaning they occur at regular intervals) or aperiodic (meaning they occur unpredictably).

if there are m periodic events and event i occurs with period Pi and requires Ci sec of CPU time to handle each event, then the load can be handled only if $\Sigma ( ( Ci / Pi ) \leq 1 )$
A real-time system that meets this criterion is said to be **schedulable**.

### *2.4.5 Policy Versus Mechanism*

none of the schedulers discussed above accept any input from user processes about scheduling decisions. As a result, the scheduler rarely makes the best choice. Sometimes a process knows of its children how important it is, and you want the important ones first.

The solution to this problem is to separate the **scheduling mechanism** from the **scheduling policy**.
What this means is that the scheduling algorithm is parameterized in some way, but the parameters can be filled in by user processes.

Suppose that the kernel uses a priority-scheduling algorithm but provides a system call by which a process can set (and change) the priorities of its children. In this way, the parent can control how its children are scheduled, even though it itself does not do the scheduling. Here the mechanism is in the kernel but policy is set by a user process.

2.4.6 Thread Scheduling
differs a lot between user-level and kernel-level threads.
kernel-level:

user-level: Since the kernel does not know that these threads even exist, it just schedules processes and the process itself schedules the threads. there are no clock interrupts so the thread may use the whole quantum, until another process is used.

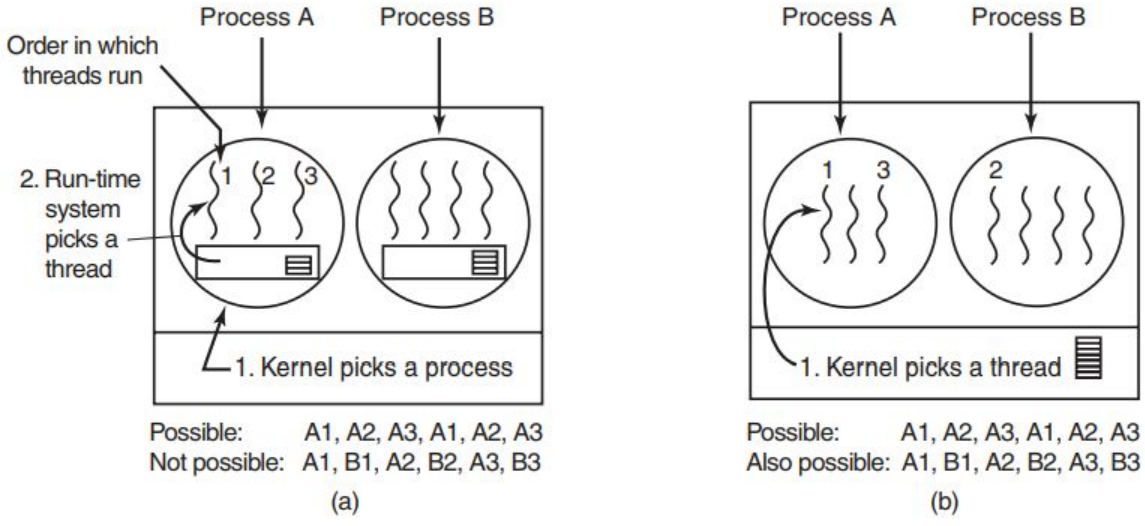



**Figure 2-44.** (a) Possible scheduling of user-level threads with a 50-msec process quantum and threads that run 5 msec per CPU burst. (b) Possible scheduling of kernel-level threads with the same characteristics as (a).

A major difference between user-level threads and kernel-level threads is the performance. Doing a thread switch with user-level threads takes a handful of machine instructions. With kernel-level threads it requires a full context switch, changing the memory map and invalidating the cache, which is several orders of magnitude slower.

Since the kernel knows that switching from a thread in process A to a thread in process B is more expensive than running a second thread in process A (due to having to change the memory map and having the memory cache spoiled), it can take this information into account when making a decision.

2.5.1 The Dining Philosophers Problem

Five philosophers are seated around a circular table. Each philosopher has a plate of spaghetti. The spaghetti is so slippery that a philosopher needs two forks to eat it. Between each pair of plates is one fork. The life of a philosopher consists of alternating periods of eating and thinking.  Can you write a program for each philosopher that does what it is supposed to do and never gets stuck?

Suppose that all five philosophers take their left forks simultaneously. None will be able to take their right forks, and there will be a deadlock.

We could easily modify the program so that after taking the left fork, the program checks to see if the right fork is available. If it is not, the philosopher puts down the left one, waits for some time, and then repeats the whole process. With a little bit of bad luck, all the philosophers could start the algorithm simultaneously, picking up their left forks, seeing that their right forks were not available, putting down their left forks, waiting, picking up their left forks again simultaneously, and so on, forever. A situation like this, in which all the programs continue to run indefinitely but fail to make any progress, is called **starvation**.

solution: a neighbour may only eat of his neighbours are not eating.

### *2.5.2 The Readers and Writers Problem*

It is acceptable to have multiple processes reading the database at the same time, but if one process is updating (writing) the database, no other processes may have access to the database, not even readers. The question is how do you program the readers and the writers?

In this solution, the first reader to get access to the database does a down on the semaphore db. Subsequent readers merely increment a counter, rc. As readers leave, they decrement the counter, and the last to leave does an up on the semaphore, allowing a blocked writer, if there is one, to get in.

Suppose that while a reader is using the database, another reader comes along. Since having two readers at the same time is not a problem, the second reader is admitted. Additional readers can also be admitted if they come along.

Now suppose a writer shows up. The writer may not be admitted to the database, since writers must have exclusive access, so the writer is suspended. Later, additional readers show up. As long as at least one reader is still active, subsequent readers are admitted. As a consequence of this strategy, as long as there is a steady supply of readers, they will all get in as soon as they arrive. The writer will be kept suspended until no reader is present. If a new reader arrives, say, every 2 sec, and each reader takes 5 sec to do its work, the writer will never get in.

To avoid this situation, the program could be written slightly differently: when a reader arrives and a writer is waiting, the reader is suspended behind the writer instead of being admitted immediately. In this way, a writer has to wait for readers that were active when it arrived to finish but does not have to wait for readers that came along after it. The disadvantage of this solution is that it achieves less concurrency and thus lower performance.

## *3 MEMORY MANAGEMENT*

To paraphrase Parkinson's Law, ''Programs expand to fill the memory available to hold them.''

people discovered the concept of a **memory hierarchy**, in which computers have a few megabytes of very fast, expensive, volatile cache memory, a few gigabytes of medium-speed, medium-priced, volatile main memory, and a few terabytes of slow, cheap, nonvolatile magnetic or solid-state disk storage, not to mention removable storage, such as DVDs and USB sticks. It is the job of the operating system to abstract this hierarchy into a useful model and then manage the abstraction.

The part of the operating system that manages (part of) the memory hierarchy is called the **memory manager**. Its job is to efficiently manage memory: keep track of which parts of memory are in use, allocate memory to processes when they need it, and deallocate it when they are done.

### *3.1 NO MEMORY ABSTRACTION*

The simplest memory abstraction is to have no abstraction at all. Early mainframe computers (before 1960), early minicomputers (before 1970), and early personal computers (before 1980) had no memory abstraction. Every program simply saw the physical memory.

The model of memory presented to the programmer was simply physical memory, a set of addresses from 0 to some maximum, each address corresponding to a cell containing some number of bits, commonly eight.

Under these conditions, it was not possible to have two running programs in memory at the same time. If the first program wrote a new value to, say, location 2000, this would erase whatever value the second program was storing there. Nothing would work and both programs would crash almost immediately.

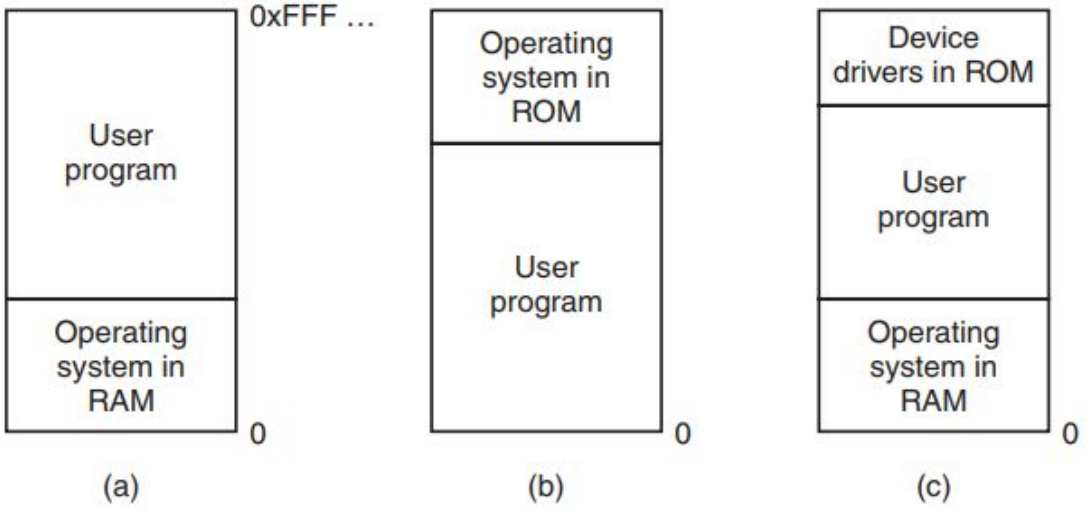



**Figure 3-1.** Three simple ways of organizing memory with an operating system and one user process. Other possibilities also exist.

Parallelism in these systems is achieved by threads.

**Running Multiple Programs Without a Memory Abstraction**

However, even with no memory abstraction, it is possible to run multiple programs at the same time. What the operating system has to do is save the entire contents of memory to a disk file, then bring in and run the next program. As long as there is only one program at a time in memory, there are no conflicts.

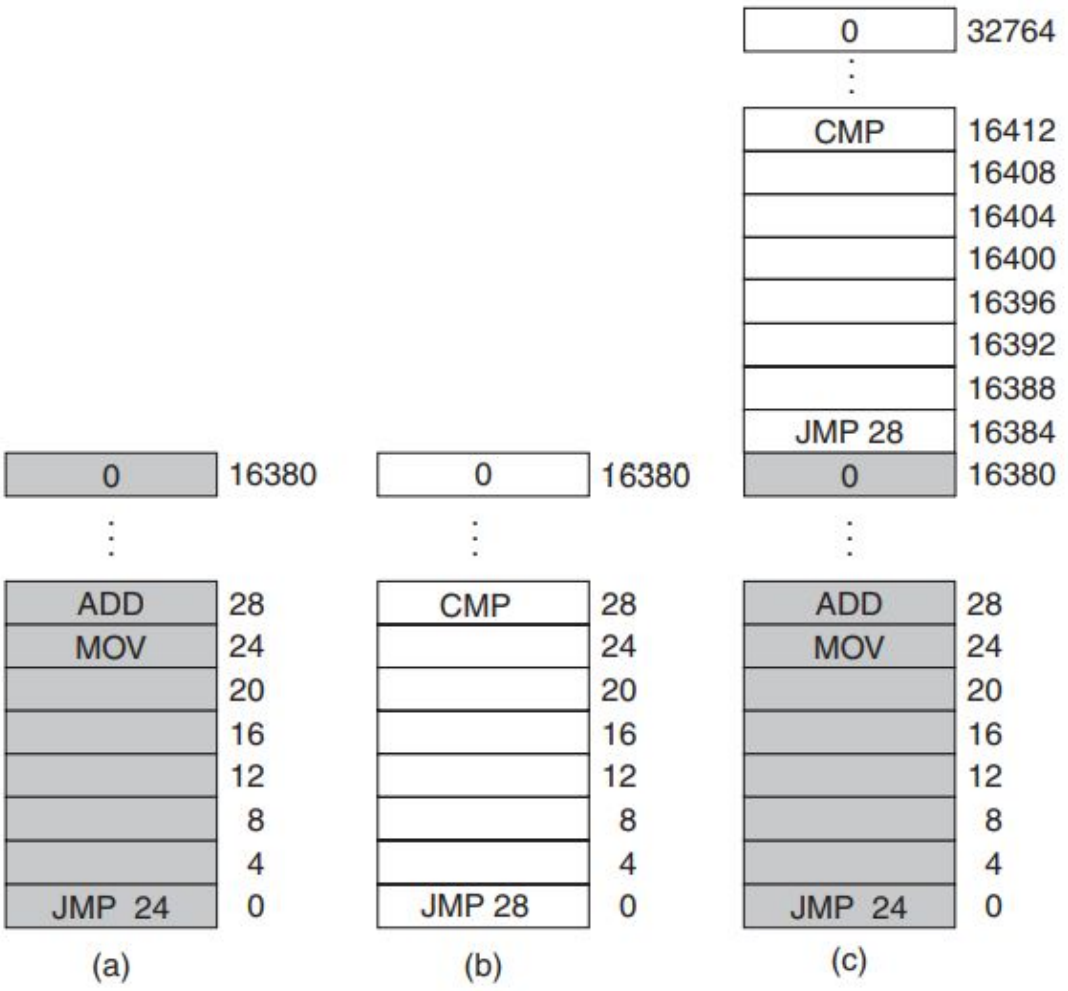



**Figure 3-2.** Illustration of the relocation problem. (a) A 16-KB program. (b) Another 16-KB program. (c) The two programs loaded consecutively into memory.

The core problem here is that the two programs both reference absolute physical memory. program 2 jumps to block 28, which is program 1 again and thus fails to execute properly.

We can target this problem by **static relocation**. When a program was loaded at address 16384, all the addresses were added with 16384. jmp 28 became jmp 16412, which works.

## *3.2 A MEMORY ABSTRACTION: ADDRESS SPACES*

Two problems with no abstraction:

- if user programs can address every byte of memory, they can easily trash the operating system, intentionally or by accident, bringing the system to a grinding halt
- it is difficult to have multiple programs running at once

### *3.2.1 The Notion of an Address Space*

Two problems have to be solved to allow multiple applications to be in memory at the same time without interfering with each other: protection and relocation.

An **address space** is the set of addresses that a process can use to address memory. Each process has its own address space, independent of those belonging to other processes (except in some special circumstances where processes want to share their address spaces).

Address 28 in one program means a different physical location than address 28 in another program.

Base and Limit Registers
This simple solution uses a particularly simple version of **dynamic relocation**.
What it does is map each process' address space onto a different part of physical memory in a simple way.
process' address space onto a different part of physical memory in a simple way. The classical solution, which was used on machines ranging from the CDC 6600 , is to equip each CPU with two special hardware registers, usually called the **base** and **limit** registers.

When these registers are used, programs are loaded into consecutive memory locations wherever there is room and without relocation during loading. When a process is run, the base register is loaded with the physical address where its program begins in memory and the limit register is loaded with the length of the program.

the first program is run are 0 and 16,384, respectively. The values used when the second program is run are 16,384 and 32,768, respectively. If a third 16-KB program were loaded directly above the second one and run, the base and limit registers would be 32,768 and 16,384.

Every time a process references memory, either to fetch an instruction or read or write a data word, the CPU hardware automatically adds the base value to the address generated by the process before sending the address out on the memory bus.

Simultaneously, it checks whether the address offered is equal to or greater than the value in the limit register, in which case a fault is generated and the access is aborted.

A disadvantage of relocation using base and limit registers is the need to perform an addition and a comparison on every memory reference. Comparisons can be done fast, but additions are slow due to carry-propagation time unless special addition circuits are used.

### *3.2.2 Swapping*

Two general approaches to dealing with memory overload have been developed over the years. The simplest strategy, called **swapping**, consists of bringing in each process in its entirety, running it for a while, then putting it back on the disk.

Idle processes are mostly stored on disk, so they do not take up any memory when they are not running (although some of them wake up periodically to do their work, then go to sleep again).

The other strategy, called **virtual memory**, allows programs to run even when they are only partially in main memory.

When swapping creates multiple holes in memory, it is possible to combine them all into one big one by moving all the processes downward as far as possible. This technique is known as **memory compaction**. It is usually not done because it requires a lot of CPU time. For example, on a 16-GB machine that can copy 8 bytes in 8 nsec, it would take about 16 sec to compact all of memory.

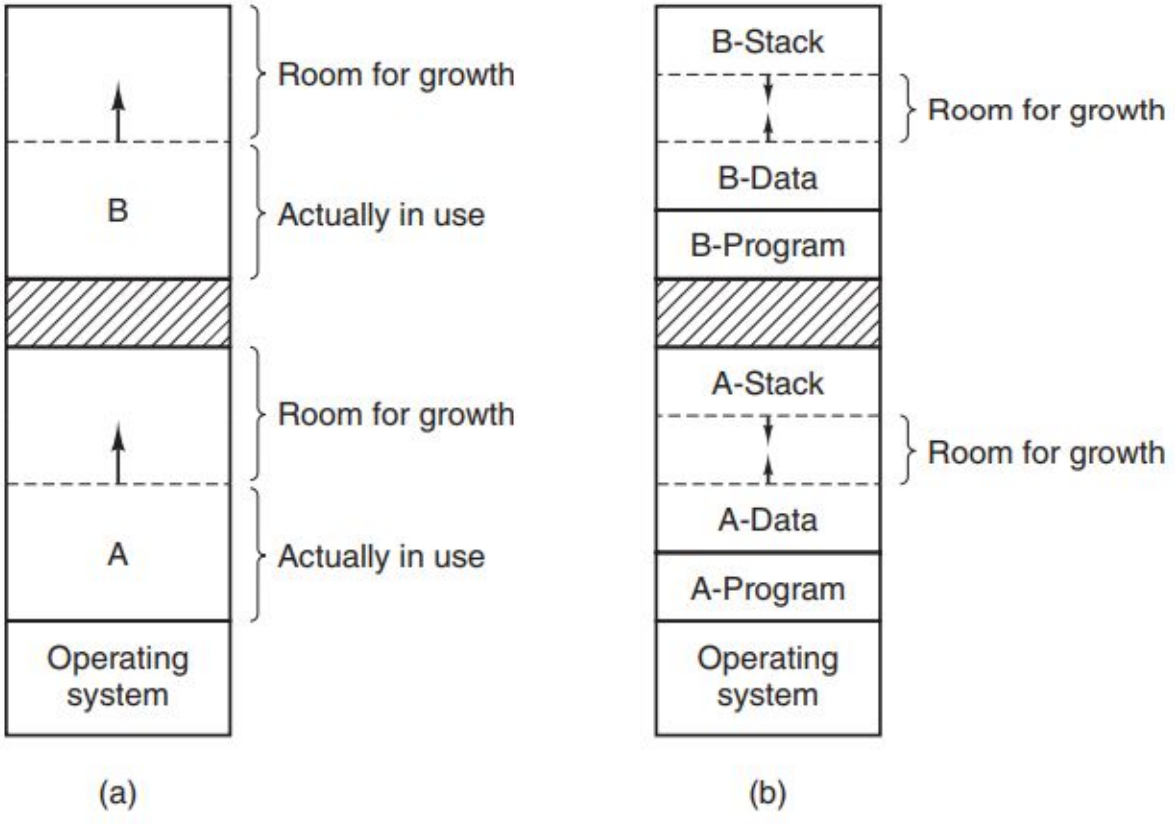



**Figure 3-5.** (a) Allocating space for a growing data segment. (b) Allocating space for a growing stack and a growing data segment.

### *3.2.3 Managing Free Memory*

**Memory Management with Bitmaps**

With a bitmap, memory is divided into allocation units as small as a few words and as large as several kilobytes. Corresponding to each allocation unit is a bit in the bitmap, which is 0 if the unit is free and 1 if it is occupied (or vice versa).

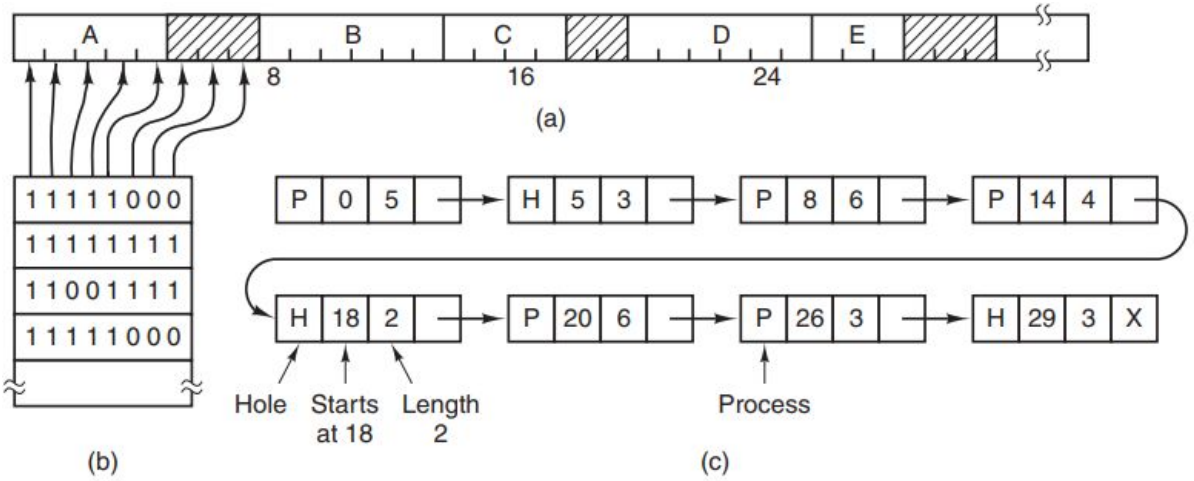



**Figure 3-6.** (a) A part of memory with five processes and three holes. The tick marks show the memory allocation units. The shaded regions (0 in the bitmap) are free. (b) The corresponding bitmap. (c) The same information as a list.

A bitmap provides a simple way to keep track of memory words in a fixed amount of memory because the size of the bitmap depends only on the size of memory and the size of the allocation unit. The main problem is that when it has been decided to bring a k-unit process into memory, the memory manager must search the bitmap to find a run of k consecutive 0 bits in the map. Searching a bitmap for a run of a given length is a slow operation (because the run may straddle word boundaries in the map); this is an argument against bitmaps.

**Memory Management with Linked Lists**

Another way of keeping track of memory is to maintain a linked list of allocated and free memory segments, where a segment either contains a process or is an empty hole between two processes


**Figure 3-7.** Four neighbor combinations for the terminating process, *X*.

When the processes and holes are kept on a list sorted by address, several algorithms can be used to allocate memory for a created process.

We assume that the memory manager knows how much memory to allocate. The simplest algorithm is **first fit**. The memory manager scans along the list of segments until it finds a hole that is big enough. The hole is then broken up into two pieces, one for the process and one for the unused memory, except in the statistically unlikely case of an exact fit. First fit is a fast algorithm because it searches as little as possible.

A minor variation of first fit is next fit. It works the same way as first fit, except that it keeps track of where it is whenever it finds a suitable hole. The next time it is called to find a hole, it starts searching the list from the place where it left off last time, instead of always at the beginning, as first fit does.

Another well-known and widely used algorithm is **best fit**. Best fit searches the entire list, from beginning to end, and takes the smallest hole that is adequate. Rather than breaking up a big hole that might be needed later, best fit tries to find a hole that is close to the actual size needed, to best match the request and the available holes.

Best fit is slower than first fit because it must search the entire list every time it is called. Somewhat surprisingly, it also results in more wasted memory than first fit or next fit because it tends to fill up memory with tiny, useless holes. First fit generates larger holes on the average.

To get around the problem of breaking up nearly exact matches into a process and a tiny hole, one could think about **worst fit**, that is, always take the largest available hole, so that the new hole will be big enough to be useful. Simulation has shown that worst fit is not a very good idea either.

Yet another allocation algorithm is quick fit, which maintains separate lists for some of the more common sizes requested. For example, it might have a table with n entries, in which the first entry is a pointer to the head of a list of 4-KB holes, the second entry is a pointer to a list of 8-KB holes, the third entry a pointer to 12-KB holes, and so on. Holes of, say, 21 KB, could be put either on the 20-KB list or on a special list of odd-sized holes. With quick fit, finding a hole of the required size is extremely fast, but it has the same disadvantage as all schemes that sort by hole size, namely, when a process terminates or is swapped out, finding its neighbors to see if a merge with them is possible is quite expensive. If merging is not done, memory will quickly fragment into a large number of small holes into which no processes fit.

### *3.3 VIRTUAL MEMORY*

While base and limit registers can be used to create the abstraction of address spaces, there is another problem that has to be solved: managing bloatware. While memory sizes are increasing rapidly, software sizes are increasing much faster.

The method that was devised (Fotheringham, 1961) has come to be known as virtual memory. The basic idea behind **virtual memory** is that each program has its own address space, which is broken up into chunks called pages. Each page is a contiguous range of addresses. These pages are mapped onto physical memory, but not all pages have to be in physical memory at the same time to run the program.

#### *3.3.1 Paging*

Most virtual memory systems use a technique called **paging.**

Program-generated addresses are called **virtual addresses** and form the **virtual address space**. On computers without virtual memory, the virtual address is put directly onto the memory bus and causes the physical memory word with the same address to be read or written. When virtual memory is used, the virtual addresses do not go directly to the memory bus. Instead, they go to an MMU that maps the virtual addresses onto the physical memory addresses.

The virtual address space consists of fixed-size units called pages. The corresponding units in the physical memory are called page frames. The pages and page frames are generally the same size.

When the program tries to access address 0, for example, using the instruction MOV REG,0 virtual address 0 is sent to the MMU. The MMU sees that this virtual address falls in page 0 (0 to 4095), which according to its mapping is page frame 2 (8192 to 12287). It thus transforms the address to 8192 and outputs address 8192 onto the bus. The memory knows nothing at all about the MMU and just sees a request for reading or writing address 8192, which it honors. Thus, the MMU has effectively mapped all virtual addresses between 0 and 4095 onto physical addresses 8192 to 12287.

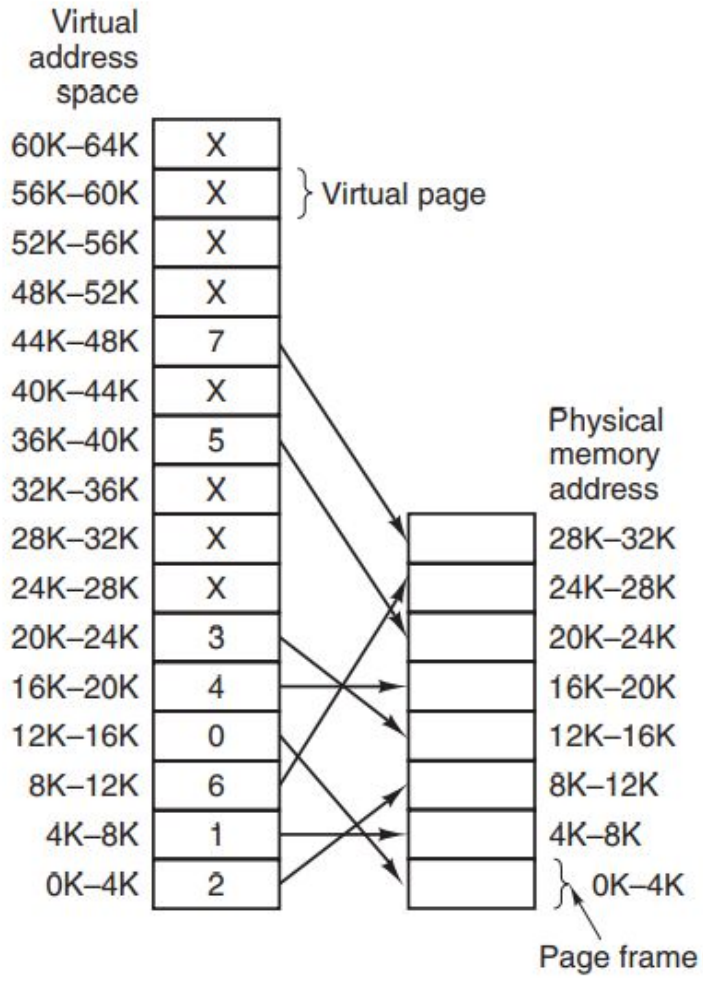



**Figure 3-9.** The relation between virtual addresses and physical memory addresses is given by the **page table**. Every page begins on a multiple of 4096 and ends 4095 addresses higher, so 4K–8K really means 4096–8191 and 8K to 12K means 8192–12287.

In the actual hardware, a **Present/absent bit** keeps track of which pages are physically present in memory.

What happens if the program references an unmapped address, for example, by using the instruction MOV REG,32780 which is byte 12 within virtual page 8 (starting at 32768)? The MMU notices that the page is unmapped (indicated by a cross in the figure) and causes the CPU to trap to the operating system. This trap is called a page fault. The operating system picks a little-used page frame and writes its contents back to the disk (if it is not already there). It then fetches (also from the disk) the page that was just referenced into the page frame just freed, changes the map, and restarts the trapped instruction.

The page number is used as an index into the page table, yielding the number of the page frame corresponding to that virtual page. If the Present/absent bit is 0, a trap to the operating system is caused. If the bit is 1, the page frame number found in the page table is copied to the high-order 3 bits of the output register, along with the 12-bit offset, which is copied unmodified from the incoming virtual address. Together they form a 15-bit physical address. The output register is then put onto the memory bus as the physical memory address.

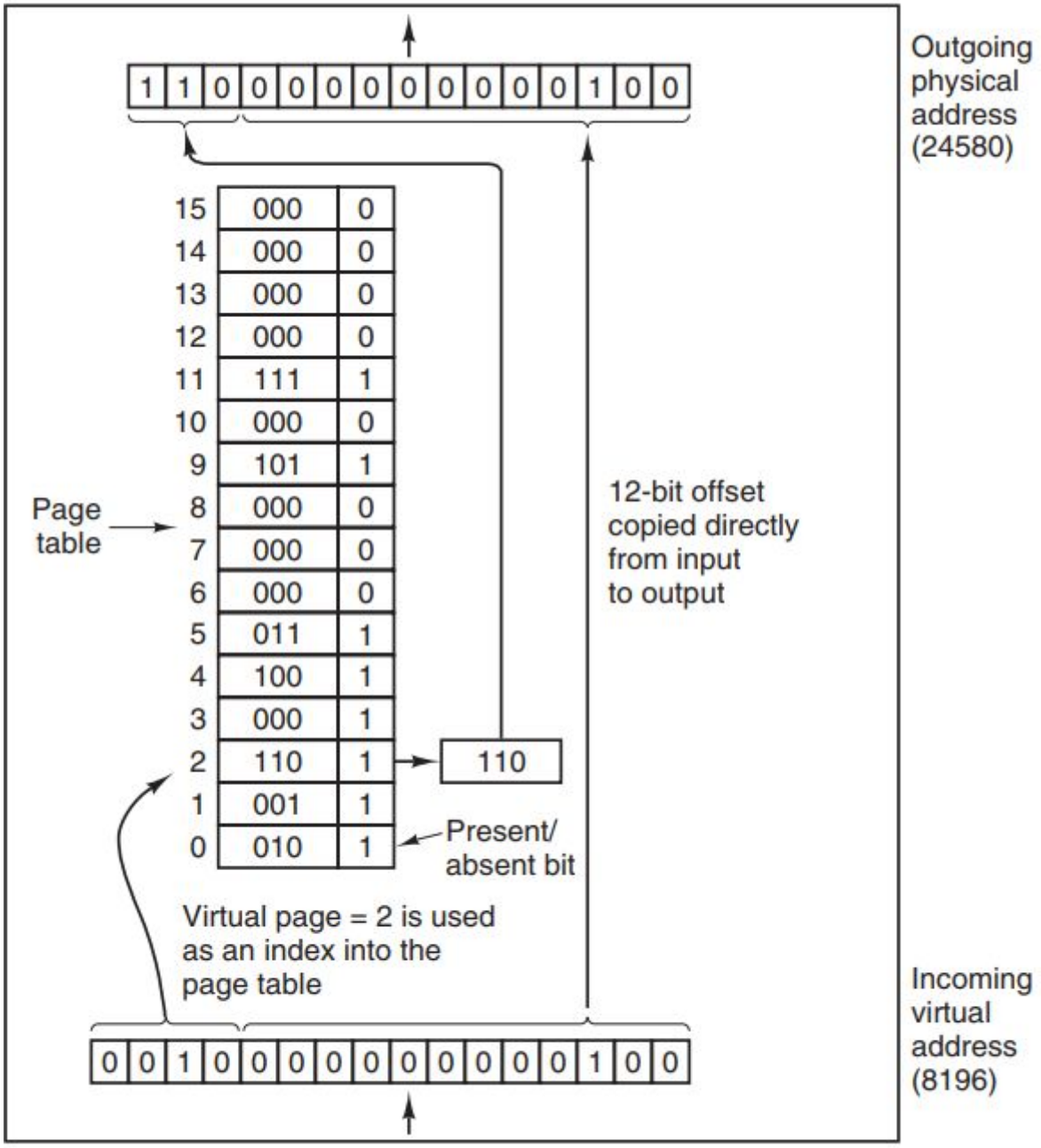



**Figure 3-10.** The internal operation of the MMU with 16 4-KB pages.

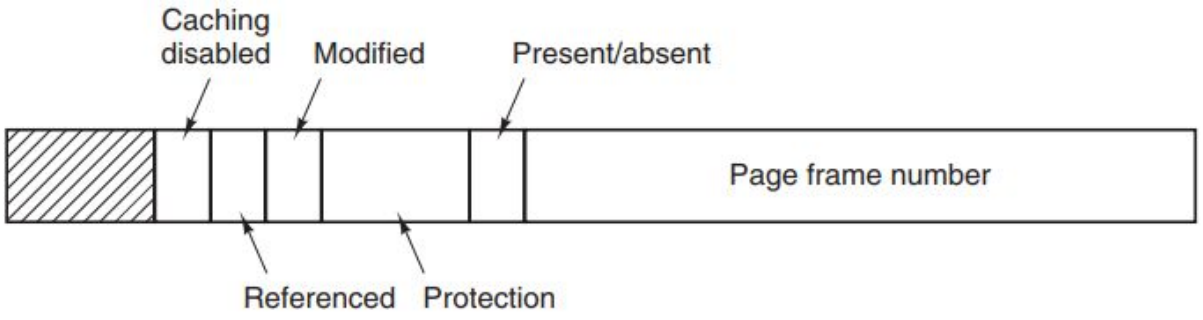
**Structure of a Page Table Entry**

The size varies from computer to computer, but 32 bits is a common size. The most important field is the Page frame number. After all, the goal of the page mapping is to output this value. Next to it we have the Present/absent bit. If this bit is 1, the entry is valid and can be used. If it is 0, the virtual page to which the entry belongs is not currently in memory. Accessing a page table entry with this bit set to 0 causes a page fault.

The Protection bits tell what kinds of access are permitted. In the simplest form, this field contains 1 bit, with 0 for read/write and 1 for read only. A more sophisticated arrangement is having 3 bits, one bit each for enabling reading, writing, and executing the page.

The Modified and Referenced bits keep track of page usage. When a page is written to, the hardware automatically sets the Modified bit. This bit is of value when the operating system decides to reclaim a page frame. If the page in it has been modified (i.e., is ‘‘dirty’’), it must be written back to the disk. If it has not been modified (i.e., is ‘‘clean’’), it can just be abandoned, since the disk copy is still valid. The bit is sometimes called the dirty bit, since it reflects the page’s state. The Referenced bit is set whenever a page is referenced, either for reading or for writing. Its value is used to help the operating system choose a page to evict when a page fault occurs. Pages that are not being used are far better candidates than pages that are, and this bit plays an important role in several of the page replacement algorithms that we will study later in this chapter.

Finally, the last bit allows caching to be disabled for the page. This feature is important for pages that map onto device registers rather than memory. If the operating system is sitting in a tight loop waiting for some I/O device to respond to a command it was just given, it is essential that the hardware keep fetching the word from the device, and not use an old cached copy. With this bit, caching can be turned off.




Before getting into more implementation issues, it is worth pointing out again that what virtual memory fundamentally does is create a new abstraction—the address space—which is an abstraction of physical memory, just as a process is an abstraction of the physical processor (CPU). Virtual memory can be implemented by breaking the virtual address space up into pages, and mapping each one onto some page frame of physical memory or having

it (temporarily) unmapped. Thus this section is bassically about an abstraction created by the operating system and how that abstraction is managed.

### *3.3.3 Speeding Up Paging*

In any paging system, two major issues must be faced:
1. The mapping from virtual address to physical address must be fast.
2. If the virtual address space is large, the page table will be large.

Translation Lookaside Buffers

Having a 1 byte instruction would take 1 memory reference.
With mapping and paging you would have at least 2. The performance is at least halved!

only a small fraction of the page table entries are heavily read; the rest are barely used at all. The solution that has been devised is to equip computers with a small hardware device for mapping virtual addresses to physical addresses without going through the page table. The device, called a TLB (Translation Lookaside Buffer) or sometimes an associative memory

It is usually inside the MMU and consists of a small number of entries, eight in this example, but rarely more than 256. Each entry contains information about one page, including the virtual page number, a bit that is set when the page is modified, the protection code (read/write/execute permissions), and the physical page frame in which the page is located. These fields have a one-to-one correspondence with the fields in the page table, except for the virtual page number, which is not needed in the page table. Another bit indicates whether the entry is valid (i.e., in use) or not.

When a virtual address is presented to the MMU for translation, the hardware first checks to see if its virtual page number is present in the TLB by comparing it to all the entries simultaneously

Doing so requires special hardware, which all MMUs with TLBs have. If a valid match is found and the access does not violate the protection bits, the page frame is taken directly from the TLB, without going to the page table. If the virtual page number is present in the TLB but the instruction is trying to write on a read-only page, a protection fault is generated.

The interesting case is what happens when the virtual page number is not in the TLB. The MMU detects the miss and does an ordinary page table lookup. It then evicts one of the entries from the TLB and replaces it with the page table entry just looked up. Thus if that page is used again soon, the second time it will result in a TLB hit rather than a miss. When an entry is purged from the TLB, the modified bit is copied back into the page table entry in memory. The other values are already there, except the reference bit. When the TLB is loaded from the page table, all the fields are taken from memory.

Software TLB Management

do nearly all of this page management in software. On these machines, the TLB entries are explicitly loaded by the operating system. When a TLB miss occurs, instead of the MMU going to the page tables to find and fetch the needed page reference, it just generates a TLB fault and tosses the problem into the lap of the operating system. The system must find the page, remove an entry from the TLB, enter the new one, and restart the instruction that faulted. And, of course, all of this must be done in a handful of instructions because TLB misses occur much more frequently than page faults.

Surprisingly enough, if the TLB is moderately large (say, 64 entries) to reduce the miss rate, software management of the TLB turns out to be acceptably efficient. The main gain here is a much simpler MMU, which frees up a considerable amount of area on the CPU chip for caches and other features that can improve performance.

To reduce TLB misses, sometimes the operating system can use its intuition to figure out which pages are likely to be used next and to preload entries for them in the TLB. For example, when a client process sends a message to a server process on the same machine, it is very likely that the server will have to run soon. Knowing this, while processing the trap to do the send, the system can also check to see where the server's code, data, and stack pages are and map them in before they get a chance to cause TLB faults.

faults can be reduced by maintaining a large (e.g., 4-KB) software cache of TLB entries in a fixed location whose page is always kept in the TLB. By first checking the software cache, the operating system can substantially reduce TLB misses.

A soft miss occurs when the page referenced is not in the TLB, but is in memory. All that is needed here is for the TLB to be updated. No disk I/O is needed.

In contrast, a hard miss occurs when the page itself is not in memory (and of course, also not in the TLB). A disk access is required to bring in the page, which can take sev eral milliseconds, depending on the disk being used. A hard miss is easily a million times slower than a soft miss. Looking up the mapping in the page table hierarchy is known as a page table walk

### *3.3.4 Page Tables for Large Memories*

TLBs can be used to speed up virtual-to-physical address translation over the original page-table-in-memory scheme. But that is not the only problem we have to tackle. Another problem is how to deal with very large virtual address spaces. Below we will discuss two ways of dealing with them.

Multilevel Page Tables
As a first approach, consider the use of a multilevel page table. A simple example is shown in Fig. 3-13. In Fig. 3-13(a) we have a 32-bit virtual address that is partitioned into a 10-bit PT1 field, a 10-bit PT2 field, and a 12-bit Offset field. Since offsets are 12 bits, pages are 4 KB, and there are a total of 220 of them.

The secret to the multilevel page table method is to avoid keeping all the page tables in memory all the time. In particular, those that are not needed should not be kept around. Suppose, for example, that a process needs 12 megabytes: the bottom 4 megabytes of memory for program text, the next 4 megabytes for data, and the top 4 megabytes for the stack. In between the top of the data and the bottom of the stack is a gigantic hole that is not used.

The two-level page table system of Fig. 3-13 can be expanded to three, four, or more levels. Additional levels give more flexibility. For instance, Intel's 32 bit 80386 processor (launched in 1985) was able to address up to 4-GB of memory, using a two-level page table that consisted of a page directory whose entries pointed to page tables, which, in turn, pointed to the actual 4-KB page frames. Both the page directory and the page tables each contained 1024 entries, giving a total of 210 × 210 × 212 = 232 addressable bytes, as desired.

**Inverted Page Tables**

An alternative to ever-increasing levels in a paging hierarchy is known as inverted page tables.
In this design, there is one entry per page frame in real memory, rather than one entry per page of virtual address space.

Although inverted page tables save lots of space, at least when the virtual address space is much larger than the physical memory, they hav e a serious downside: virtual-to-physical translation becomes much harder. When process n references virtual page p, the hardware can no longer find the physical page by using p as an index into the page table. Instead, it must search the entire inverted page table for an entry (n, p). Furthermore, this search must be done on every memory reference, not just on page faults. Searching a 256K table on every memory reference is not the way to make your machine blindingly fast

The way out of this dilemma is to make use of the TLB. If the TLB can hold all of the heavily used pages, translation can happen just as fast as with regular page tables. On a TLB miss, however, the inverted page table has to be searched in software. One feasible way to accomplish this search is to have a hash table hashed on the virtual address. All the virtual pages currently in memory that have the same hash value are chained together.

Inverted page tables are common on 64-bit machines because even with a very large page size, the number of page table entries is gigantic. For example, with 4-MB pages and 64-bit virtual addresses, 242 page table entries are needed.

# 4 File systems

we have three essential requirements for long-term information storage:

1. It must be possible to store a very large amount of information.
2. The information must survive the termination of the process using it.
3. Multiple processes must be able to access the information at once.

it is sufficient to think of a disk as a linear sequence of fixed-size blocks and supporting two operations:

1. Read block k.
2. Write block k

**Files** are logical units of information created by processes.

Processes can read existing files and create new ones if need be. Information stored in files must be **persistent**, that is, not be affected by process creation and termination.

### 4.1.1 File Naming

Many operating systems support two-part file names, with the two parts separated by a period, as in prog.c. The part following the period is called the **file extension** and usually indicates something about the file.

### 4.1.2 File Structure

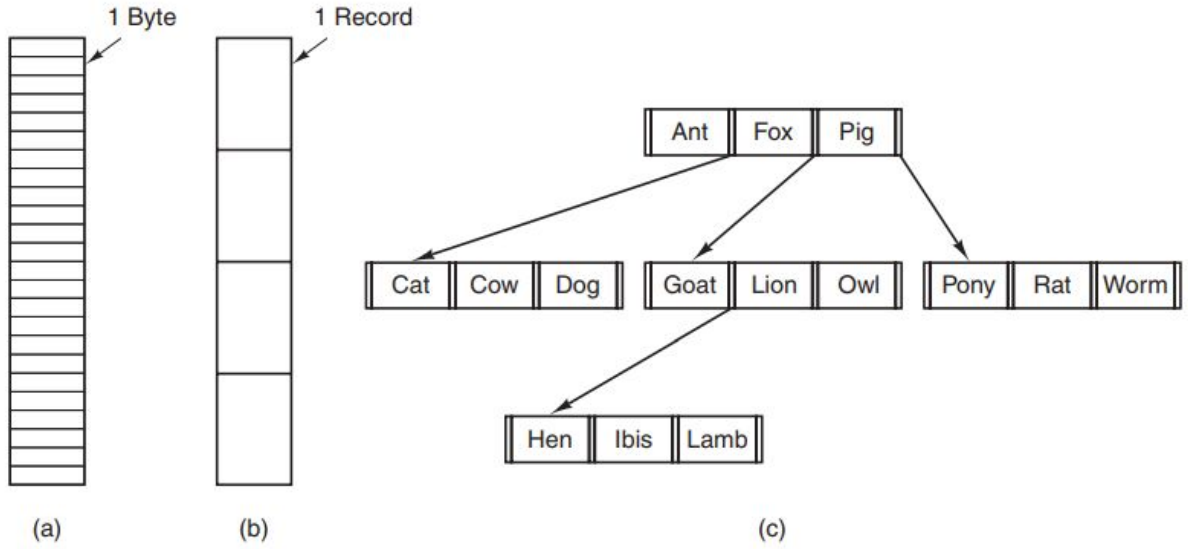



**Figure 4-2.** Three kinds of files. (a) Byte sequence. (b) Record sequence. (c) Tree.

a) file is a unstructured sequence of bytes. the operating system does not know or care what is in the file. All it sees are bytes. Any meaning must be imposed by user-level programs.

Having the operating system regard files as nothing more than byte sequences provides the maximum amount of flexibility. User programs can put anything they want in their files and name them any way that they find convenient. The operating system does not help, but it also does not get in the way.

b) In this model, a file is a sequence of fixed-length records, each with some internal structure. Central to the idea of a file being a sequence of records is the idea that the read operation returns one record and the write operation overwrites or appends one record.

c) In this organization, a file consists of a tree of records, not necessarily all the same length, each containing a key field in a fixed position in the record. The tree is sorted on the key field, to allow rapid searching for a particular key. The basic operation here is not to get the ‘‘next’’ record, although that is also possible, but to get the record with a specific key.

### 4.1.3 File Types

**Regular files** are the ones that contain user information.

**Directories** are system files for maintaining the structure of the file system.

**Character special files** are related to input/output and used to model serial I/O devices, such as terminals, printers, and networks.

**Block special files** are used to model disks.

Regular files are generally either ASCII files or binary files. ASCII files consist of lines of text.

The great advantage of ASCII files is that they can be displayed and printed as is, and they can be edited with any text editor. Furthermore, if large numbers of programs use ASCII files for input and output, it is easy to connect the output of one program to the input of another, as in shell pipelines.

Other files are binary, which just means that they are not ASCII files. Listing them on the printer gives an incomprehensible listing full of random junk. Usually, they have some internal structure known to programs that use them.

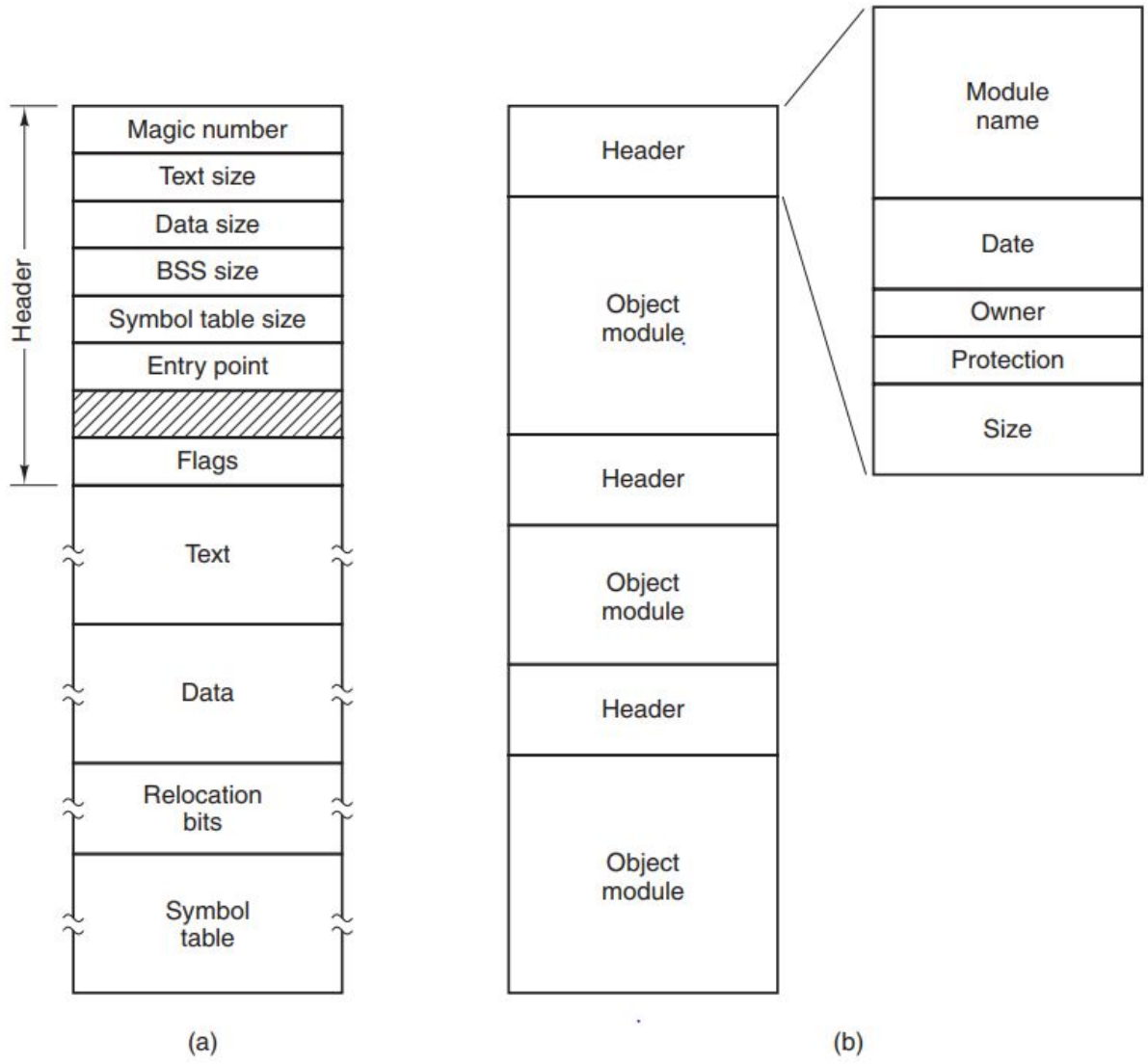



**Figure 4-3.** (a) An executable file. (b) An archive.

For example, in Fig. 4-3(a) we see a simple executable binary file taken from an early version of UNIX.

The header starts with a so-called **magic number**, identifying the file as an executable file (to prevent the accidental execution of a file not in this format)

Every operating system must recognize at least one file type: its own executable file; some recognize more.

### 4.1.4 File Access

Early operating systems provided only one kind of file access: **sequential access**. In these systems, a process could read all the bytes or records in a file in order, starting at the beginning, but could not skip around and read them out of order.

Files whose bytes or records can be read in any order are called **random-access files**.

### 4.1.5 File Attributes

Every file has a name and its data. In addition, all operating systems associate other information with each file, for example, the date and time the file was last modified and the file's size. We will call these extra items the **file's attributes**. Some people call them **metadata**.

The first four attributes relate to the **file's protection** and tell who may access it and who may not.

The flags are bits or short fields that control or enable some specific property. Hidden files, for example, do not appear in listings of all the files. The archive flag is a bit that keeps track of whether the file has been backed up recently.

The record-length, key-position, and key-length fields are only present in files whose records can be looked up using a key. They provide the information required to find the keys.

The various times keep track of when the file was created, most recently accessed, and most recently modified. These are useful for a variety of purposes.

The current size tells how big the file is at present. Some old mainframe operating systems required the maximum size to be specified when the file was created, in order to let the operating system reserve the maximum amount of storage in advance

| Attribute | Meaning |
|---|---|
| Protection | Who can access the file and in what way |
| Password | Password needed to access the file |
| Creator | ID of the person who created the file |
| Owner | Current owner |
| Read-only flag | 0 for read/write; 1 for read only |
| Hidden flag | 0 for normal; 1 for do not display in listings |
| System flag | 0 for normal files; 1 for system file |
| Archive flag | 0 for has been backed up; 1 for needs to be backed up |
| ASCII/binary flag | 0 for ASCII file; 1 for binary file |
| Random access flag | 0 for sequential access only; 1 for random access |
| Temporary flag | 0 for normal; 1 for delete file on process exit |
| Lock flags | 0 for unlocked; nonzero for locked |
| Record length | Number of bytes in a record |
| Key position | Offset of the key within each record |
| Key length | Number of bytes in the key field |
| Creation time | Date and time the file was created |
| Time of last access | Date and time the file was last accessed |
| Time of last change | Date and time the file was last changed |
| Current size | Number of bytes in the file |
| Maximum size | Number of bytes the file may grow to |

**Figure 4-4.** Some possible file attributes.

4.1.6 File Operations

Create: file is created without data

delete: file is deleted

open: file needs to be opened before using the file

Close: file is closed after

Read: data is read from file

write: data is written to the file

append:  restricted form of write, can only write at the end of the file

seek:  For random-access files, a method is needed to specify from where to take the data.

Get attributes: Processes often need to read file attributes to do their work.

. Set attributes.:  some attributes are settable and can be changed after the file has been created

Rename:  It frequently happens that a user needs to change the name of an existing file. This system call makes that possible.

### 4.2 DIRECTORIES

To keep track of files, file systems normally have **directories** or **folders**, which are themselves files.

#### 4.2.1 Single-Level Directory Systems

The simplest form of directory system is having one directory containing all the files. Sometimes it is called the **root directory**, but since it is the only one, the name does not matter much

#### 4.2.2 Hierarchical Directory Systems

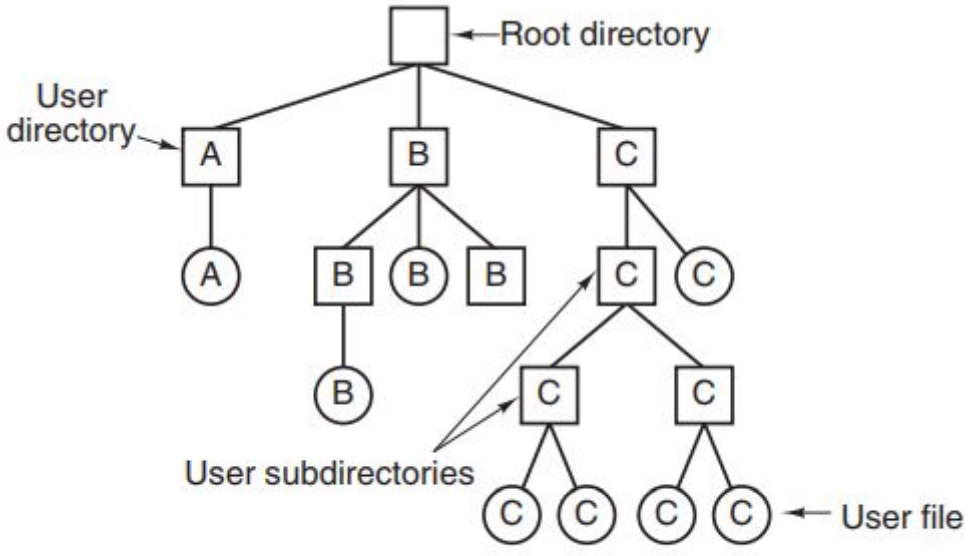



**Figure 4-7.** A hierarchical directory system.

The ability for users to create an arbitrary number of subdirectories provides a powerful structuring tool for users to organize their work. For this reason, nearly all modern file systems are organized in this manner.

4.2.3 Path Names

When the file system is organized as a directory tree, some way is needed for specifying file names. Two different methods are commonly used. In the first method, each file is given an absolute path name consisting of the path from the root directory to the file.

No matter which character is used, if the first character of the path name is the separator, then the path is absolute.

The other kind of name is the relative path name. This is used in conjunction with the concept of the working directory (also called the current directory). A user can designate one directory as the current working directory, in which case all path names not beginning at the root directory are taken relative to the working directory.

Most operating systems that support a hierarchical directory system have two special entries in every directory, ‘‘.’’ and ‘‘..’’, generally pronounced ‘‘dot’’ and ‘‘dotdot.’’ Dot refers to the current directory; dotdot refers to its parent (except in the root directory, where it refers to itself).

**4.2.4 Directory Operations**

Create. A directory is created.

Delete. A directory is deleted.

Opendir. Directories can be read.

Closedir. When a directory has been read, it should be closed to free up internal table space.

Readdir. This call returns the next entry in an open directory. Formerly, it was possible to read directories using the usual read system call, but that approach has the disadvantage of forcing the programmer to know and deal with the internal structure of directories. In contrast, readdir always returns one entry in a standard format, no matter which of the possible directory structures is being used.

Rename. In many respects, directories are just like files and can be renamed the same way files can be.

Link. Linking is a technique that allows a file to appear in more than one directory.

Unlink. A directory entry is removed.

## 4.3 FILE-SYSTEM IMPLEMENTATION

### 4.3.1 File-System Layout

File systems are stored on disks. Most disks can be divided up into one or more partitions, with independent file systems on each partition. Sector 0 of the disk is called the **MBR** (Master Boot Record) and is used to boot the computer.

When the computer is booted, the BIOS reads in and executes the MBR. The first thing the MBR program does is locate the active partition, read in its first block, which is called the **boot block**, and execute it.

The first one is the superblock. It contains all the key parameters about the file system and is read into memory when the computer is booted or the file system is first touched. Typical information in the superblock includes a magic number to identify the file-system type, the number of blocks in the file system, and other key administrative information.

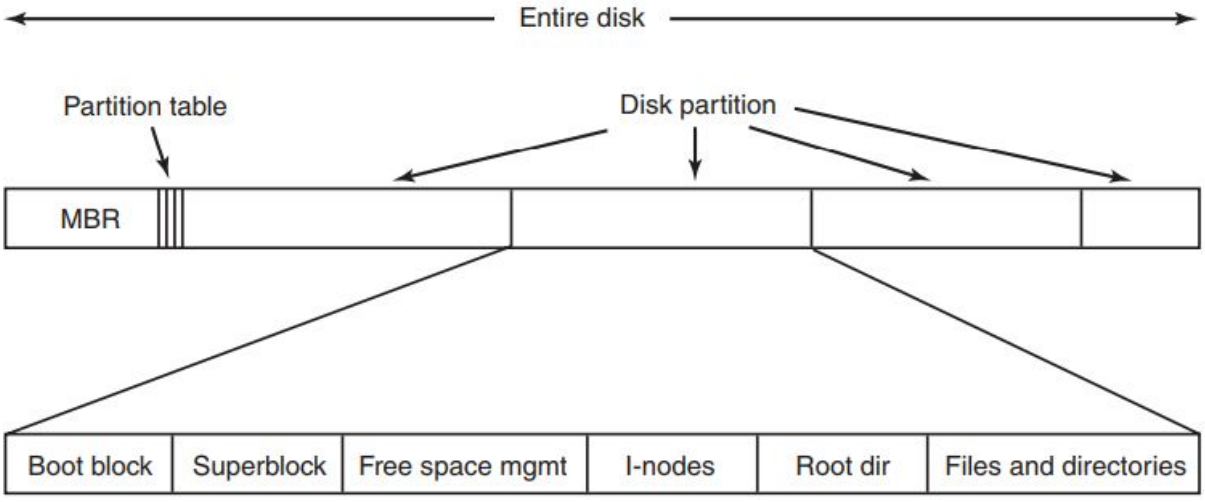


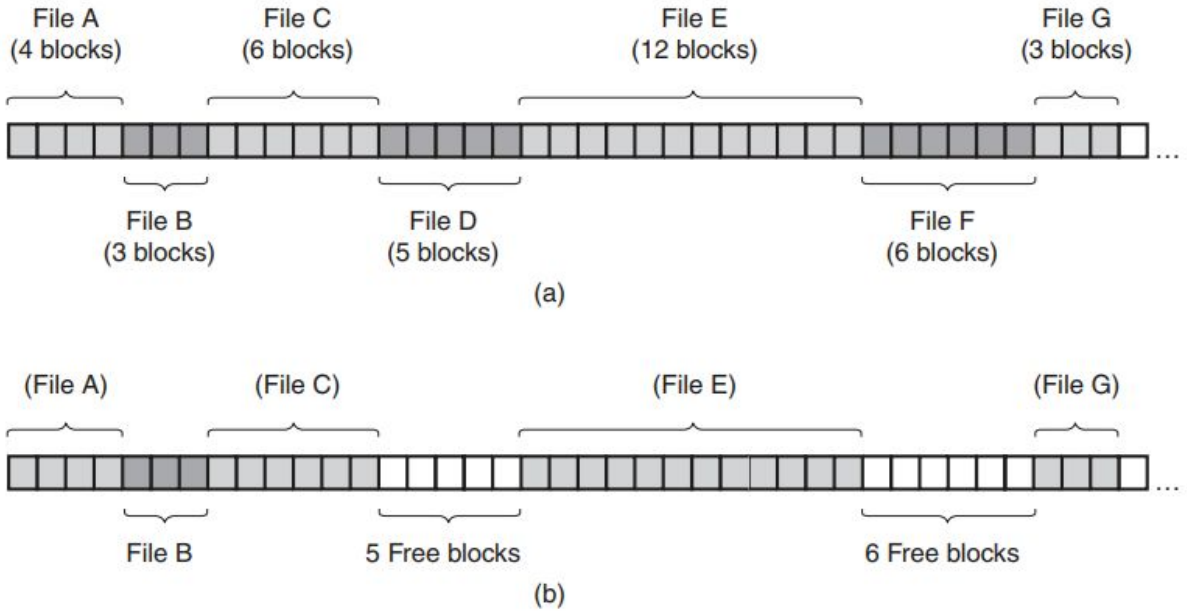

**Figure 4-9.** A possible file-system layout.

### 4.3.2 Implementing Files

Probably the most important issue in implementing file storage is keeping track of which disk blocks go with which file.

#### Contiguous Allocation

The simplest allocation scheme is to store each file as a contiguous run of disk blocks. Thus on a disk with 1-KB blocks, a 50-KB file would be allocated 50 consecutive blocks. With 2-KB blocks, it would be allocated 25 consecutive blocks.

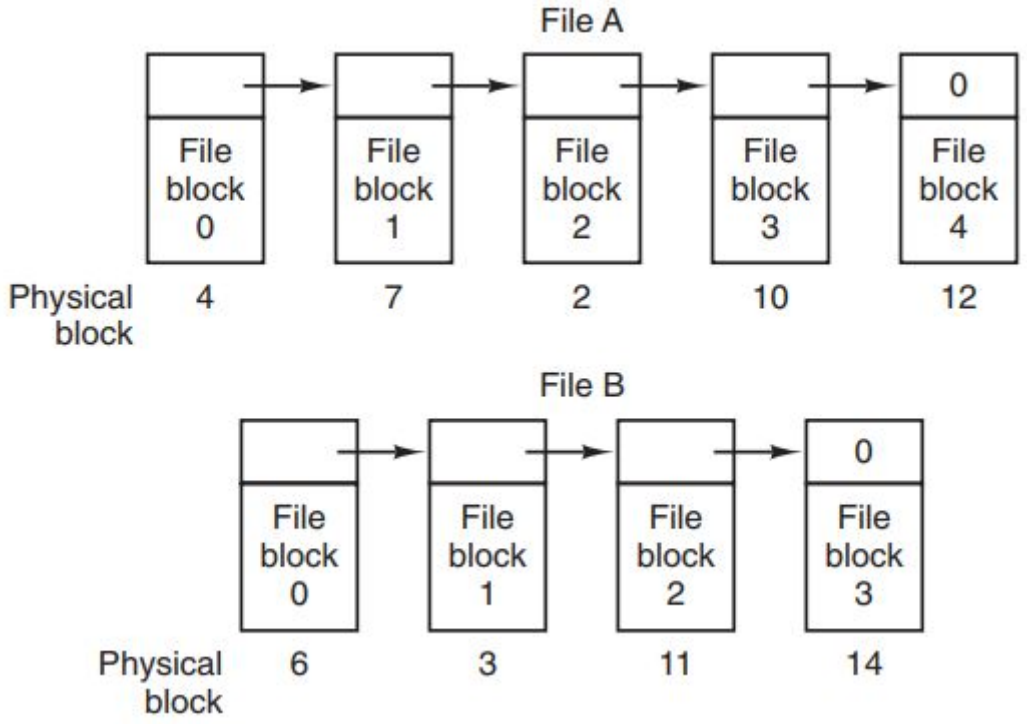



Unfortunately, contiguous allocation also has a very serious drawback: over the course of time, the disk becomes fragmented.

Initially, this fragmentation is not a problem, since each new file can be written at the end of disk, following the previous one. However, eventually the disk will fill up and it will become necessary to either compact the disk, which is prohibitively expensive, or to reuse the free space in the holes. Reusing the space requires maintaining a list of holes, which is doable. However, when a new file is to be created, it is necessary to know its final size in order to choose a hole of the correct size to place it in.

However, there is one situation in which contiguous allocation is feasible and, in fact, still used: on CD-ROMs. Here all the file sizes are known in advance and will never change during subsequent use of the CD-ROM file system.

**Linked-List Allocation**



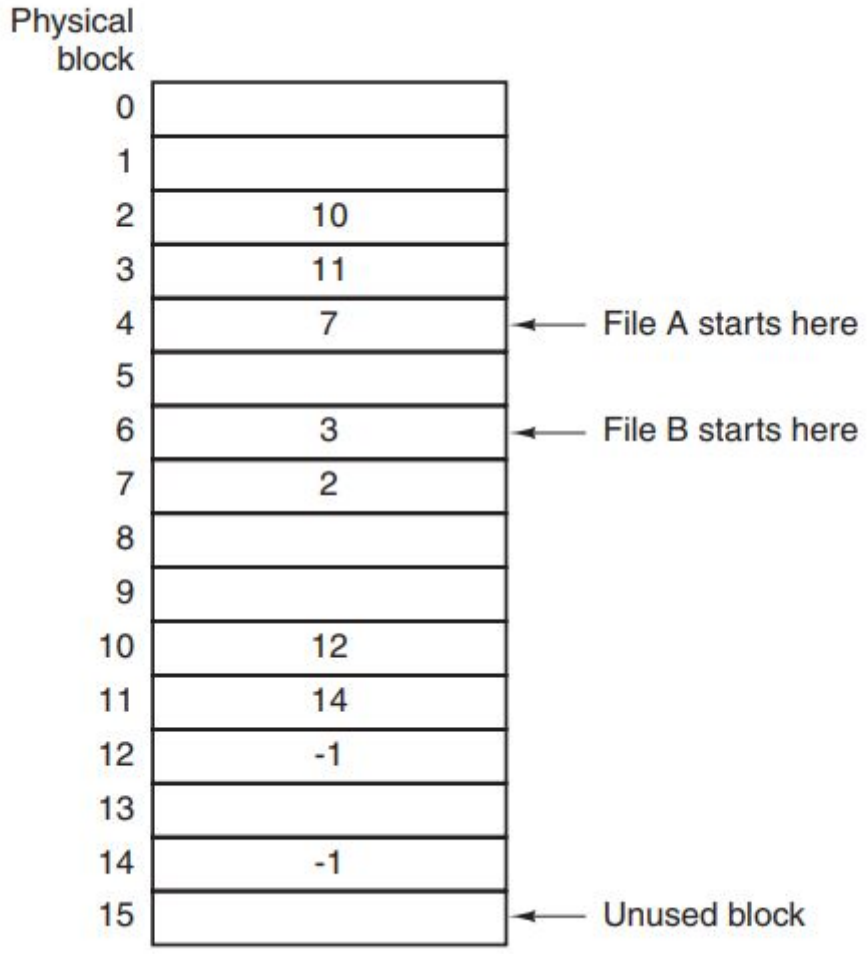

Unlike contiguous allocation, every disk block can be used in this method. No space is lost to disk fragmentation

lso, it is sufficient for the directory entry to merely store the disk address of the first block. The rest can be found starting there.

On the other hand, although reading a file sequentially is straightforward, random access is extremely slow. To get to block n, the operating system has to start at the beginning and read the n − 1 blocks prior to it, one at a time. Clearly, doing so many reads will be painfully slow.

**Linked-List Allocation Using a Table in Memory**

Both disadvantages of the linked-list allocation can be eliminated by taking the pointer word from each disk block and putting it in a table in memory.




**I- nodes**

Our last method for keeping track of which blocks belong to which file is to associate with each file a data structure called an i-node (index-node), which lists the attributes and disk addresses of the file's blocks.

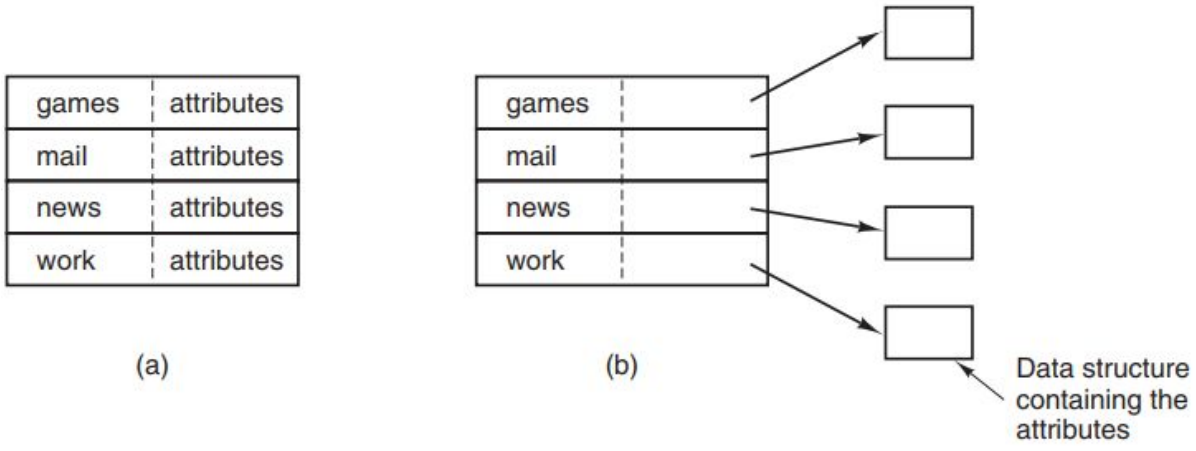
The big advantage of this scheme over linked files using an in-memory table is that the i-node need be in memory only when the corresponding file is open.

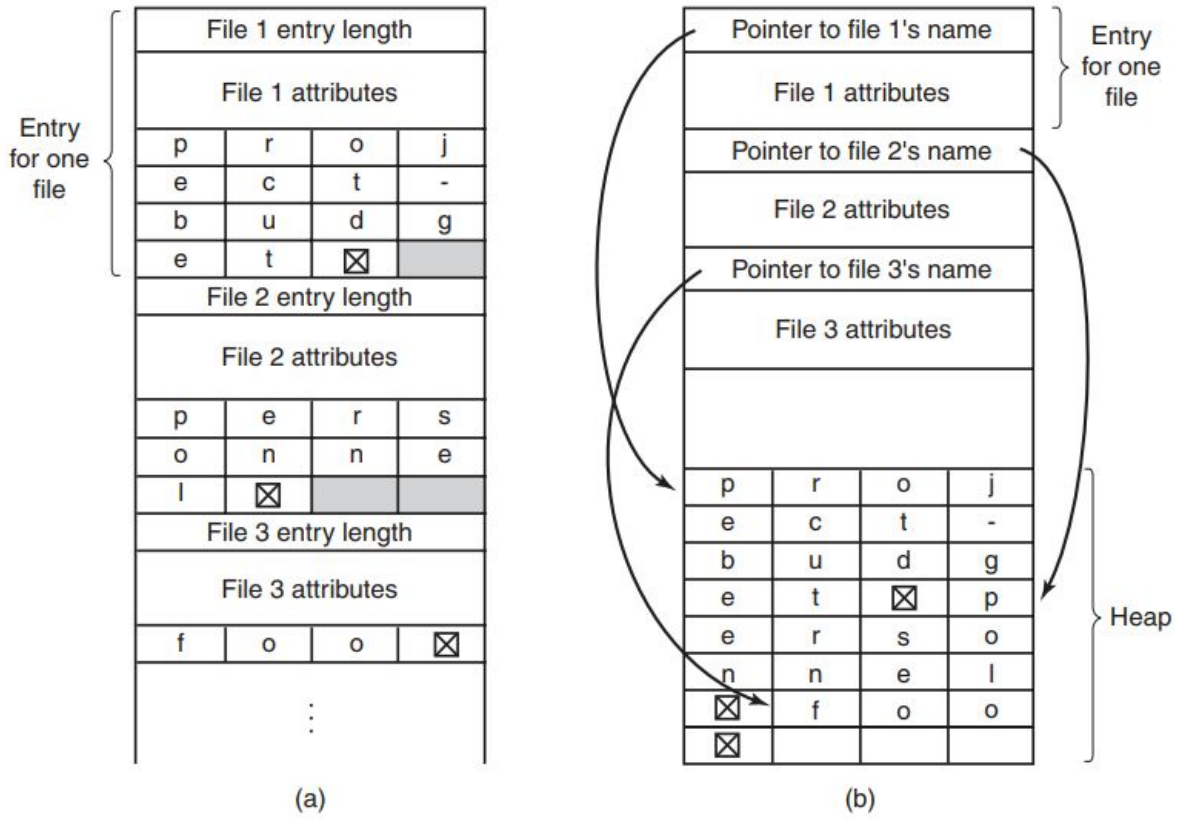
4.3.3 Implementing Directories

Before a file can be read, it must be opened. When a file is opened, the operating system uses the path name supplied by the user to locate the directory entry on the disk. The directory entry provides the information needed to find the disk blocks. Depending on the system, this information may be the disk address of the entire file (with contiguous allocation), the number of the first block (both linked-list schemes), or the number of the i-node. In all cases, the main function of the directory system is to map the ASCII name of the file onto the information needed to locate the data.

Every file system maintains various file attributes, such as each file's owner and creation time, and they must be stored somewhere. One obvious possibility is to store them directly in the directory entry. Some systems do precisely that.

For systems that use i-nodes, another possibility for storing the attributes is in the i-nodes, rather than in the directory entries. In that case, the directory entry can be shorter: just a file name and an i-node number.






A disadvantage of this method is that when a file is removed, a variable-sized gap is introduced into the directory into which the next file to be entered may not fit. This problem is essentially the same one we saw with contiguous disk files, only now compacting the directory is feasible because it is entirely in memory.

### 4.3.4 Shared Files

When several users are working together on a project, they often need to share files. As a result, it is often convenient for a shared file to appear simultaneously in different directories belonging to different users.

The connection between B's directory and the shared file is called a link. The file system itself is now a Directed Acyclic Graph, or DAG, rather than a tree.

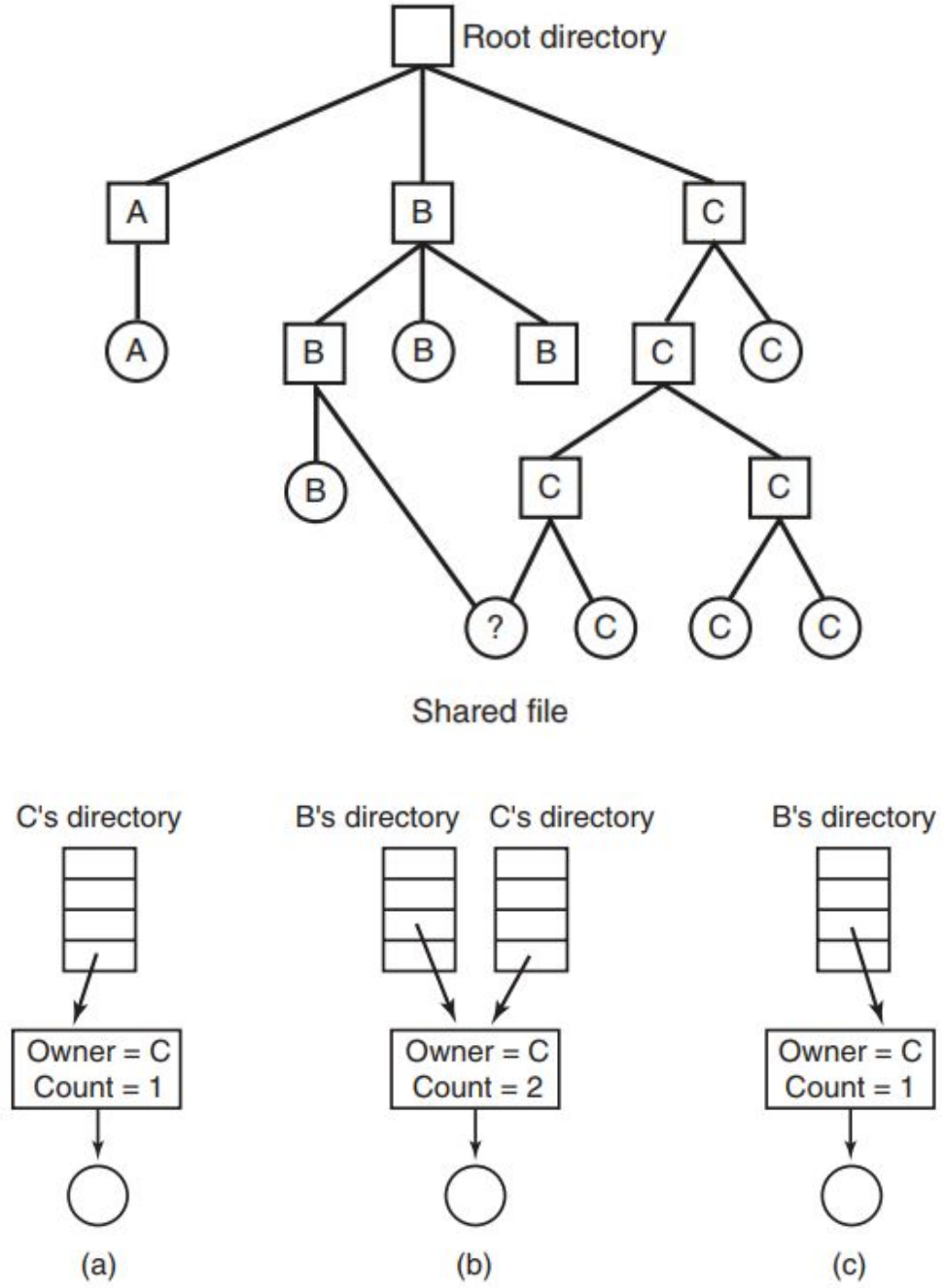



**Figure 4-17.** (a) Situation prior to linking. (b) After the link is created. (c) After the original owner removes the file.