OPERATING SYSTEMS

STUDY GUIDE

Organized & Prepared By

Sharafat Ibn Mollah Mosharraf

12th Batch (05-06)
Dept. of Computer Science & Engineering
University of Dhaka

Table of Contents

Chapter 2
Processes, Threads, IPC & Scheduling

Roadmap and Concepts in brief

- In this chapter, we'll learn how operating systems manage processes.
- First of all, what is a process?

  A process is just an executing program, including the *current* values of the program counter, registers and variables.
- So what does an OS has to do with processes?

  An OS should know *when* and *how* to create, terminate and manage processes.

  - Why would an OS bother managing processes? Simply loading a program into RAM from HDD and executing it should not require any special management, should it?

    Well, no. But when multiple processes are executed simultaneously (i.e., multitasked), they need to be managed so that when one process blocks, another one can continue.

    - Why multitasking is needed, anyway?

      Because,
      1. When a process blocks on I/O, the CPU remains idle, as I/O device data transfer is much slower than data transfer between CPU and RAM.
      2. A running process should be suspended when a higher-priority process arrives.
      3. A user can do multiple tasks at the same time.

  - So when should an OS create processes?
    1. **System initialization** – daemons are created
    2. **Execution of a *process creation system call* by running process** – e.g. `fork` and `execve`.
    3. **A user request to create a new process** – by executing a program file.
    4. **Initiation of a batch job** – when the OS decides that it has the resources to run another job, it creates a new process and runs the next job from the input queue in it.

  - When should an OS terminate processes?
    1. **Normal exit (voluntary)** – by calling `exit` in UNIX or `ExitProcess` in Windows and passing it a value of 0.
    2. **Error exit (voluntary)** – if the program discovers a fatal error (e.g. no filename has been provided by user to a file copying program when it is executed), it exits by calling exit and passing it a non-zero value.
    3. **Fatal error (involuntary)** – an error caused by the process (often due to program bugs) which it cannot handle; e.g. executing illegal instruction, dividing by zero etc.
    4. **Killed by another process (involuntary)** – by calling `kill` in UNIX or `TerminateProcess` in Windows.

  - How does an OS manage processes?

    Whenever a process blocks, the OS suspends it and executes another process. When the second process finishes or blocks, the OS resumes the first process.

    - Well, then there exists some states of a process and some situations when the process switches from one state to another. Explain these states and situations. In other words, explain the mechanism of state transition.

      See *Process States* [*Theory 2.6*].

    - What happens to the context (e.g. the current register values etc.) of the blocked process when another process is executed? Are they lost?

      Nope! The process context is saved when context switch occurs.

      - Great! But *where* and *how*?

        See *PCB and others…* [*Theory 2.7 & 2.8*]

- What are threads?

  Different segments of code of a single process which run concurrently are called threads.

- Why do we need multiple threads instead of multiple processes?

  Threads share a single address space and can thus operate on the same resource. Multiple processes have different address spaces and thus *cannot* operate on the same resource.

- So how does operating system manage threads?

  In any of the following three ways:

  1. **User-level threads:**
     a. Each process has a thread table in its address space.
     b. Thread switching is done entirely in user-space using a run-time system.
  2. **Kernel-level threads:**
     a. A single thread table resides in kernel space.
     b. Thread switching is controlled by kernel.
  3. **Hybrid implementation (scheduler activation):**
     a. One or more threads in user-space connect to one or more threads in kernel-space.
     b. The kernel-level threads are scheduled by kernel and the user-level threads by processes.

- Okay. You've explained how operating systems manage processes. But what about process synchronization and interprocess communication? i.e., how does the operating system manages processes running in parallel so that they can share some data and operate on it without any synchronization problems?

  To avoid race conditions and thereby solve the critical section problem, we can do any of the followings:

  1. **By disabling interrupts:** Each process would disable all interrupts just after entering its critical region and re-enable them just before leaving it.
  2. **By using mutual exclusion:** For example:
     a. **Mutual exclusion with busy waiting**
        i. **Lock Variable:**

           A single, shared (lock) variable is used for mutual exclusion. If the value of the variable is 0, it means that no process is in its critical region; and if it is 1, it means that some process is in its critical region.

        ii. **Strict Alternation:**

           A variable *turn* keeps track of whose turn it is to enter the critical region.

        iii. **Peterson's Solution:**

           There are two functions – `enter_region()` and `leave_region()`, a `turn` variable and an `interested` array. (The `turn` variable and the `interested` array combinely acts as a *lock* variable).

        iv. **The TSL (Test and Set Lock) instruction:**

           The lock variable is controlled by hardware using the TSL instruction.

     b. **Mutual exclusion with `sleep` and `wakeup`**
        i. **Semaphore / Counting Semaphore**

           A counting semaphore is a counter for *a set of* available resources, rather than a locked/unlocked flag of a *single* resource.

        ii. **Mutex / Lock / Mutex Lock / Binary Semaphore**

           It is a simplified version of the semaphore, having only two possible values – 0 or 1.

        iii. **Monitor**

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

➢ Okay. Final question! *When* and *how* does an operating system schedule processes or threads?

An operating system schedules processes when a process is created, terminates, switches from *running* to *ready* or *waiting/blocked* state, or switches from *waiting/blocked* to *ready* state.

Scheduling algorithms:

1. **First-Come First-Served (FCFS) [*Non-preemptive*] [*Applies to: Batch Systems*]**
   - ✓ Processes are assigned the CPU in the order they request it.
   - ✓ There is a single queue of *ready* processes. When the running process blocks, the first process on the queue is run next. When a blocked process becomes ready, like a newly arrived job, it is put on the end of the queue.
2. **Shortest Job First (SJF) [*Non-preemptive*] [*Applies to: Batch Systems*]**

   When several equally important jobs are sitting in the input queue waiting to be started, the scheduler picks the shortest job first (assuming run times are known in advance).
3. **Shortest Remaining Time Next (SRTN) /**
   Shortest Remaining Time First (SRTF) /
   Shortest Time to Completion First (STCF) /
   Preemptive Shortest Job First (Preemptive SJF)
   [*Preemptive*] [*Applies to: Batch Systems*]

   When a new job arrives, its total time is compared to the current process' remaining time. If the new job needs less time to finish than the current process, the current process is suspended (and placed at the end of the queue) and the new job is started.
4. **Round-Robin Scheduling (RR Scheduling)**
   [*Preemptive*] [*Applies to: Both Batch and Interactive Systems*]

   When the process uses up its quantum, it is put on the *end* of the ready list.
5. **Priority Scheduling**
   [*Preemptive / Non-preemptive*] [*Applies to: Both Batch and Interactive Systems*]

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

   Priority scheduling can be either *preemptive* or *nonpreemptive*. A preemptive priority scheduling algorithm will preempt the CPU if the priority of the newly arrived process is higher than the priority of the currently running process. A nonpreemptive priority scheduling algorithm will simply put the new process at the head of the ready queue.
6. **Multiple / Multilevel Queue Scheduling (MQS / MQ Scheduling)**
   [*Fixed-Priority Preemptive (Commonly)*] [*Applies to: All the systems*]

   A multilevel queue scheduling algorithm partitions the ready queue into several separate queues. The processes are permanently assigned to one queue. Each queue has its own scheduling algorithm.

   There must be scheduling among the queues, which is commonly implemented as fixed-priority preemptive scheduling.
7. **Multilevel Feedback-Queue Scheduling (MFQS / MFQ Scheduling) /**
   Exponential Queue
   [*Fixed-Priority Preemptive (Commonly)*] [*Applies to: All the systems*]

   The *Multilevel Feedback-Queue Scheduling* algorithm allows a process to move between queues. The idea is to separate processes according to the *characteristics of their CPU bursts*. If a process uses too much CPU time, it will be moved to a lower-priority queue.
8. **Guaranteed Scheduling**
   [*Preemptive*] [*Applies to: Interactive systems*]

   Makes real promises to the users about performance and then fulfills them. One promise that is realistic to make and easy to fulfill is this: If there are *n* users logged in while you are working, you will receive about 1/*n* of the CPU power.

9. **Lottery Scheduling**
[*Preemptive*] [*Applies to: Interactive systems*]

The basic idea is to give processes lottery tickets for various system resources, such as CPU time. Whenever a scheduling decision has to be made, a lottery ticket is chosen at random, and the process holding that ticket gets the resource.

10. **Fair-Share Scheduling**
[*Preemptive*] [*Applies to: Interactive systems*]

Some systems take into account *who* owns a process before scheduling it. In this model, each user is allocated some fraction of the CPU and the scheduler picks processes in such a way as to enforce it. Thus 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.

11. **Rate Monotonic Scheduling (RMS)**
[*Preemptive*] [*Applies to: Real-time systems*]

It is a static fixed-priority scheduling where priorities are assigned based on the period of each process. The shorter the period, the higher the priority.

12. **Earliest Deadline First Scheduling (EDFS)**
[*Preemptive*] [*Applies to: Real-time systems*]

The algorithm runs the first process on the list, the one with the closest deadline. Whenever a new process becomes ready, the system checks to see if its deadline occurs before that of the currently running process. If so, the new process preempts the current one.

Acronyms

EDFS	-	Earliest Deadline First Scheduling (*Concept 2.26*)
FCFS	-	First-Come First-Served (*Concept 2.25*)
IPC	-	Interprocess Communication (*Concept 2.14*)
LWP	-	Lightweight Processes (i.e. threads) (*Concepts 2.9 & 2.13*)
MFQS	-	Multilevel Feedback-Queue Scheduling (*Concept 2.25*)
MQS	-	Multiple / Multilevel Queue Scheduling (*Concept 2.25*)
PCB	-	Process Control Block (*Concept 2.7*)
PID	-	Process ID
RMS	-	Rate Monotonic Scheduling (*Concept 2.26*)
RR	-	Round-Robin (Scheduling) (*Concept 2.25*)
SJF	-	Shortest Job First (*Concept 2.25*)
SRTF	-	Shortest Remaining Time First (*Concept 2.25*)
SRTN	-	Shortest Remaining Time Next (*Concept 2.25*)
TSL	-	Test and Lock (*Concept 2.15*)
VAS	-	Virtual Address Space

Glossary

2.1	**Clock interrupt time / CPU Burst length / Run time / Time quanta** The amount of time a process uses a processor before it is no longer ready.
2.2	**Console** *(compare **terminal**, **shell**)* A *console* is a keyboard/display unit *locally* connected to a machine.
2.3	**Context Switch** A *context switch* is the process of storing and restoring the state (context) of a CPU such that multiple processes can share a single CPU resource.

2.4 Daemon

Processes that stay in the background to handle some activity such as email, web pages, printing etc. are called *daemons* (in UNIX systems). Typically daemons have names that end with the letter 'd'; for example, `syslogd`, the daemon that handles the system log, or `sshd`, which handles incoming SSH (Secure Shell) connections.

In Windows systems, daemons are called *services*.

2.5 Kernel Trapping

Sending an interrupt to the kernel is called *kernel trapping*. Execution of a system call can be implemented by trapping the kernel.

2.6 Shell *(compare **console**, **terminal**)*

An operating system *shell* is a piece of software which provides access to the services of a kernel.

Operating system shells generally fall into one of two categories: *command line* and *graphical*. Command line shells provide a command line interface (CLI) to the operating system, while graphical shells provide a graphical user interface (GUI). In either category the primary purpose of the shell is to *invoke* or *launch* another program; however, shells frequently have additional capabilities such as viewing the contents of directories.

2.7 Starvation / Indefinite Blocking

A situation in which all the programs continue to run indefinitely but fail to make any progress is called *starvation*. Again, the situation in which a process remains in the ready queue indefinitely is also called *starvation*.

2.8 System Calls / APIs

The set of functions provided by the operating system to interface between the operating system and the user programs are called *system calls* or *APIs*.

2.9 Terminal *(compare **console**, **shell**)*

A *terminal* is a keyboard/display unit *remotely* connected to a (generally mainframe) machine.

Theory

2.1 Process / Sequential Process / Job / Task

A process is just an executing program, including the *current* values of the program counter, registers and variables.

Difference between a program and a process

A program is a *passive entity*, such as a file containing a list of instructions stored on disk (often called an *executable file*); whereas a process is an *active entity*, with a program counter and a set of associated resources.

A program becomes a process when an executable file is loaded into memory.

2.2 Structure of a process in memory

Informally, as mentioned earlier, a process is a program in execution. A process is more than the program code (a.k.a. **text section**). It (i.e., the process) also includes the current activity, as represented by the value of the **program counter** and the contents of the processor's registers. A process generally also includes the process **stack**, which contains temporary data (such as function parameters, return addresses, and local variables), and a **data section**, which contains global variables. A process may also include a **heap**, which is memory that is dynamically allocated during process run time. The structure of a process in memory is shown in *figure 2.2*.

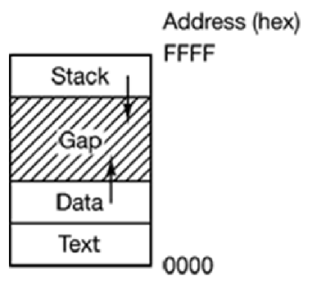


Figure 2.2: Structure of a process

The data segment grows upward and the stack grows downward, as shown in the above figure. Between them is a gap of unused address space. The stack grows into the gap automatically as needed, but expansion of the data segment is done explicitly by calling the `malloc()` library function. The expansion of data segment is called the heap.

2.3 Process Creation

There are *four* principal events that cause processes to be created:

1. **System initialization –** daemons are created
2. **Execution of a *process creation system call* by running process –** e.g. `fork` and `execve`.
3. **A user request to create a new process –** by executing a program file.
4. **Initiation of a batch job –** when the OS decides that it has the resources to run another job, it creates a new process and runs the next job from the input queue in it.

Whatever the case may be, however, a new process is created by *having an existing process execute a process creation system call*.

In UNIX, new process creation is a twp-step process:

1. `fork` – after this system call, the child process gets a duplicate, but separate copy of its parent's address space. The PID (Process ID) of the child is returned to the parent, whereas the PID of the child is assigned as 0.
2. `execve` – after this system call, the child process changes its memory image and runs 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` to accomplish redirection of standard input, standard output and standard error.

In Windows, a single system call `CreateProcess` handles both process creation and loading the correct program into the new process (as well as assigning a new PID). So, in this OS, the parent's and child's address spaces are different from the start.

2.4 Process Termination

Processes terminate usually due to one of the following conditions:

1. **Normal exit (voluntary)** – by calling `exit` in UNIX or `ExitProcess` in Windows and passing it a value of 0.
2. **Error exit (voluntary)** – if the program discovers a fatal error (e.g. no filename has been provided by user to a file copying program when it is executed), it exits by calling `exit` and passing it a non-zero value.
3. **Fatal error (involuntary)** – an error caused by the process (often due to program bugs) which it cannot handle; e.g. executing illegal instruction, dividing by zero etc.
4. **Killed by another process (involuntary)** – by calling `kill` in UNIX or `TerminateProcess` in Windows.

In UNIX, if the parent terminates, all its children are assigned the **init** process (PID 1) as their new parent.

2.5 Process Hierarchies

In UNIX, a process may create several child processes, which in turn may create further descendants, thus creating a process hierarchy.

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 (usually all active processes that were created in the current window). Individually, each process can catch the signal, ignore the signal, or take the default action, which is to be killed by the signal.

In contrast, Windows does not have any concept of a process hierarchy. All processes are equal. The only place where there is something like a process hierarchy is that when a process is created, the parent is given a special token (called a **handle)** that it can use to control the child. However, it is free to pass this token to some other process, thus invalidating the hierarchy. Processes in UNIX cannot disinherit their children.

2.6 Process States

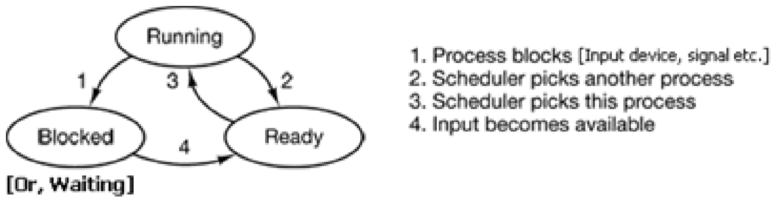

Running
Blocked
Ready
[Or, Waiting]
1
2
3
4
1. Process blocks [Input device, signal etc.]
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available

Figure 2.6: A process can be in running, blocked, or ready state. Transitions between these states are as shown.

2.7 Process Control Block (PCB) / Task Control Block / Task Struct / Proc Struct and Process Table

Each process is represented in the operating system by a process control block (PCB) – also called a *task control block*. A PCB is shown in Figure 2.3. It contains many pieces of information associated with a specific process, including these:

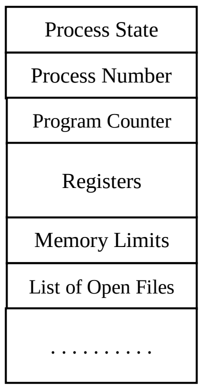


Figure 2.7: Process Control Block (PCB)

• **Process state.** The state may be new, ready, running, waiting, halted, and so on.

• **Program counter.** The counter indicates the address of the next instruction to be executed for this process.

• **CPU registers.** The registers vary in number and type, depending on the computer architecture. They include accumulators, index registers, stack pointers, and general-purpose registers, plus any condition-code information. Along with the program counter, this state information must be saved when an interrupt occurs, to allow the process to be continued correctly afterward.

• **CPU-scheduling information.** This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters.

• **Memory-management information.** This information may include such information as the value of the base and limit registers, the page tables, or the segment tables, depending on the memory system used by the operating system.

• **Accounting information.** This information includes the amount of CPU and real time used, time limits, account numbers, job or process numbers, and so on.

• **I/O status information.** This information includes the list of I/O devices allocated to the process, a list of open files, and so on.

In brief, the PCB simply serves as the repository for any information that may vary from process to process.

The OS maintains a table (an array of structures), called the ***process table***, with one entry per process, which entry is called the PCB.

2.8 How interrupts are handled [*provided for understanding how context switch is performed*]

1. Current process' program counter, program status word (i.e. status register, e.g. the flag register in x86) and possibly one or more registers are pushed onto the current (i.e. the program's) stack by the interrupt hardware.
2. The interrupt routine's address is loaded in program counter from interrupt vector table and the computer jumps to that address. [Here ends the hardware part and the software part is started from the next step.]
3. All interrupts start by saving the registers in the PCB of the current process. The registers pushed on the stack by the interrupt hardware are first saved in the PCB, and then removed from the stack.
4. The stack pointer register is set to point to a temporary stack used by the process handler.

5. The interrupt service is run.
6. After that, the scheduler is called to see who to run next.
7. The registers and memory map for the now-current process is loaded up and the process starts running.

Actions such as loading and saving the registers and setting the stack pointer (i.e., steps 3, 4 and 7) are done through Assembly Language procedures.

2.9 The Thread Model

- **Definition of thread:**
  Threads of execution are forks of a computer program into two or more concurrently running tasks which are generally independent of each other, resides in the same memory address space as the process and share the resources of the process among themselves.

- **Differences between processes and threads:**
  1. Processes are used to group resources together; threads are the entities scheduled for execution on the CPU.
  2. Processes share physical memory, disks, printers, and other resources. Threads share an address space, open files and other resources of the single process in which they are contained.

- **Lightweight Processes (LWP):** Because threads have some of the properties of processes (e.g. run in parallel independently of one-another, share some resources etc.), they are sometimes called *lightweight processes*.[1]

- **How multithreading works:** Multithreading works the same way as multitasking. The CPU switches rapidly back and forth among the threads providing the illusion that the threads are running in parallel.

- **Items shared by threads and processes:**

Per process items *(shared by all threads in a process)*	**Per thread items** *(private to each thread)*
Address space	Registers (including Program Counter)
Global variables	Stack
Open files	State (or, thread-specific data[2])
Child processes	
Pending alarms	
Signals and signal handlers	
Accounting information	

- **States of a thread:** 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.

- **Thread creation and termination (and other system calls used by threads):**

  When multithreading is present, processes normally start with a single thread present. This thread has the ability to create new threads by calling a library procedure, for example, `thread_create`[3]. A parameter to `thread_create` typically specifies the name of a procedure for the new thread to run.

[1] However, the actual definition of LWP will be provided in *Concept 2.13*.

[2] Thread-local storage (TLS) [according to Windows terminology, *thread-local storage* is the term for *thread-specific data*] is a computer programming method that uses static or global memory *local* to a thread.

This is sometimes needed because all threads in a process share the same address space. In other words, data in a static or global variable is normally always located at the same memory location, when referred to by threads from the same process. Sometimes it is desirable that two threads referring to the same static or global variable are actually referring to different memory locations, thereby making the variable thread local, a best example being the C error code variable `errno`.

[3] In POSIX.1c, it's `pthread_create`.

When a thread has finished its work, it can exit by calling a library procedure, say, `thread_exit`[4].

In some thread systems, one thread can wait for a (specific) thread to exit by calling a procedure, for example, `thread_wait`[5]. This procedure blocks the calling thread until a (specific) thread has exited.

Another common thread call is `thread_yield`[6], 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 timesharing 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.

Other calls allow one thread to wait for another thread to finish some work, for a thread to announce that it has finished some work, and so on.

➢ **Problems with implementing thread system:**

1. **Effects of `fork` system call:**
   If the parent process has multiple threads, should the child also have them?
   a. If not, the process may not function properly, since all of them may be essential.
   b. If yes, then what happens if a thread in the parent was blocked on a `read` call, say, from the keyboard? Are two threads now blocked on the keyboard, one in the parent and one in the child? When a line is typed, do both threads get a copy of it? Only the parent? Only the child? The same problem exists with open network connections.[7]

2. **Sharing the same resource:**
   a. What happens if one thread closes a file while another one is still reading from it?
   b. Suppose that one thread notices that there is too little memory and starts allocating more memory. Part way through, a thread switch occurs, and the new thread also notices that there is too little memory and also starts allocating more memory. Memory will probably be allocated twice.

2.10 Thread Usage

Why multiple *threads* and not multiple *processes*?

1. The main reason for having threads is that in many applications, multiple activities are going on at once. Some of these may block from time to time. By decomposing such an application into multiple sequential threads that run in quasi-parallel, the programming model becomes simpler. But this is precisely the argument for having multiple processes. However, with threads, we add a new element: *the ability for the parallel entities to share an address space and all of its data among themselves*. This ability is essential for certain applications, which is why having multiple processes (with their separate address spaces) will not work.

2. *Since threads do not have any resources attached to them, they are easier to create and destroy than processes*. In many systems, creating a thread goes 100 times faster than creating a process. When the number of threads needed changes dynamically and rapidly, this property is useful.

3. A third reason for having threads is also a performance argument. Threads yield no

[4] In POSIX.1c, it's `pthread_exit`.

[5] In POSIX.1c, it's `pthread_join`.

[6] In POSIX.1c, it's `sched_yield`.

[7] Some UNIX systems have chosen to have two versions of `fork`, one that duplicates all threads and another that duplicates only the thread that invoked the `fork` system call. Which of the two versions of `fork` should be used depends on the application. If `exec` is called immediately after `fork`ing, then duplicating all threads is unnecessary, as the program specified in the parameters to `exec` will replace the process. In this instance, duplicating only the calling thread is appropriate. If, however, the separate process does not call `exec` after `fork`ing, the separate process should duplicate all threads.

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*. In other words, if only processes are used, then whenever an I/O would occur, the whole process would be blocked; but in multithreading, only the thread that needs I/O would be blocked, and another thread would use the CPU for its task; hence the application speeds up.

4. The benefits of multithreading can be greatly increased in a multiprocessor architecture, where threads may be running in parallel on different processors. *A single-threaded process can only run on one CPU, no matter how many are available. Multithreading on a multi-CPU machine increases concurrency.*

Example 1: Word Processor Program

The problem

Consider what happens when the user suddenly deletes one sentence from page 1 of an 800-page document. After checking the changed page to make sure it is correct, the user now wants to make another change on page 600 and types in a command telling the word processor to go to that page (possibly by searching for a phrase occurring only there). The word processor is now forced to reformat the entire book up to page 600 on the spot because it does not know what the first line of page 600 will be until it has processed all the previous pages. There may be a substantial delay before page 600 can be displayed, leading to an unhappy user.

Solution by using threads

Suppose that the word processor is written as a two-threaded program. One thread interacts with the user and the other handles reformatting in the background. As soon as the sentence is deleted from page 1 the interactive thread tells the reformatting thread to reformat the whole book. Meanwhile, the interactive thread continues to listen to the keyboard and mouse and responds to simple commands like scrolling page 1 while the other thread is computing madly in the background. With a little luck, the reformatting will be completed before the user asks to see page 600, so it can be displayed instantly.

Another Word Processing problem and its solution using threads

Many word processors have a feature of automatically saving the entire file to disk every few minutes to protect the user against losing a day's work in the event of a program crash, system crash, or power failure. We can add a third thread that can handle the disk backups without interfering with the other two.

The problems that would arise if the Word Processing program were single-threaded

1. *Slow performance.* Whenever a disk backup started, commands from the keyboard and mouse would be ignored until the backup was finished. The user would perceive this as sluggish performance.
2. Alternatively, keyboard and mouse events could interrupt the disk backup, *allowing good performance* but *leading to a complex interrupt-driven programming model*. With three threads, the programming model is much simpler. The first thread just interacts with the user. The second thread reformats the document when told to. The third thread writes the contents of RAM to disk periodically.

What if we used multiple processes instead of multiple threads to solve this problem

Having three separate processes would not work here because all three threads need to operate on the *same* resource (i.e., the document). By having three threads instead of three processes, they share a common address space and thus all have access to the document being edited.

Example 2: Web Server

The problem

We have to design a web server which would do the following tasks:

1. Read incoming requests for web pages.
2. Checks the cache (main memory) for the requested page. If found, returns the page. If not

found, reads the page from disk and then returns the page.

Solution using multiple threads

We can use a *dispatcher* thread which reads incoming requests for web pages. Along with the dispatcher thread, we can use multiple *worker* threads which will return the requested web pages. After examining the request, the dispatcher chooses an idle (i.e., blocked) worker thread, wakes it up by moving it from blocked to ready state and hands it the request. 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. When the thread blocks on the disk operation, another thread is chosen to run, possibly the dispatcher, in order to acquire more work, or possibly another worker that is now ready to run (i.e., the worker has read the page from disk and is now ready to return it).

Note that each thread is programmed *sequentially*, in the usual way.

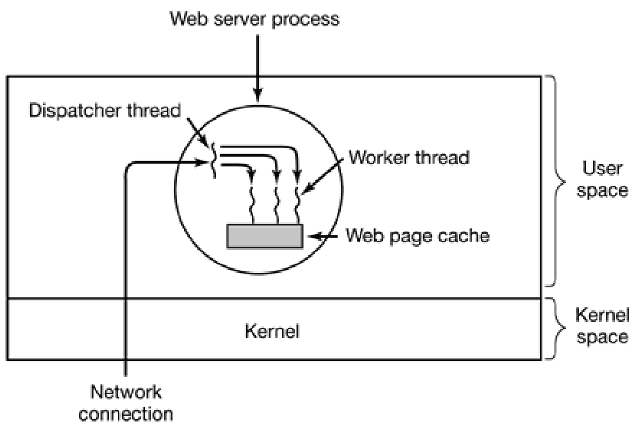


Figure 2.10: A multithreaded web server.

Solution using single thread

The main (infinite) loop of the Web server program gets a request, examines it, and carries it out to completion before getting the next request. While waiting for the disk, the server is idle and does not process any other incoming requests. If the Web server is running on a dedicated machine, as is commonly the case, the CPU is simply idle while the Web server is waiting for the disk. The net result is that many fewer *requests/sec* can be processed.

Solution using FSM (Finite-State Machine)

Suppose that threads are not available but the system designers find the performance loss due to single threading unacceptable. If a *non-blocking* 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 non-blocking 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 non-blocking 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 *finite-state machine***.**

Usefulness of threads as understood from this example

Threads make it possible to retain the idea of sequential processes that make *blocking* system 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 ease of blocking system calls, but gives up performance. The third approach achieves high performance through parallelism but uses non-blocking calls and interrupts and is thus is hard to program. These models are summarized in the following table:

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

What if we used multiple processes instead of multiple threads to solve this problem

The web page cache is a single memory resource which needs to be accessed by multiple threads residing at the same address space. If we had used multiple processes, accessing this single resource would have to be coded in a much harder way.

Example 3: Applications that must process very large amounts of data (e.g. Codecs)

The normal approach followed in such an application is to read in a block of data, process it, and then write it out again. The problem here is that if only blocking system calls are available, the process blocks while data are coming in and data are going out. Having the CPU go idle when there is lots of computing to do is clearly wasteful and should be avoided if possible.

Threads offer a solution. The process could be structured with an input thread, a processing thread, and an output thread. The input thread reads data into an input buffer. The processing thread takes data out of the input buffer, processes them, and puts the results in an output buffer. The output buffer writes these results back to disk. In this way, input, output, and processing can all be going on at the same time. Of course, this model only works if a system call blocks only the calling thread, not the entire process.

2.11 Implementing threads in user space (User-level thread package)

Organization of threads in user space

- The thread package is entirely in user space. The kernel knows nothing about them.
- The threads run on top of a run-time system (called a ***thread library***), which is a collection of procedures that manage threads. We've seen four of these already: *thread_create*, *thread_exit*, *thread_wait*, and *thread_yield*, but usually there are more.
- When threads are managed in user space, each process needs its own private **thread table** to keep track of the threads in that process. The thread table is managed by the runtime system. 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.

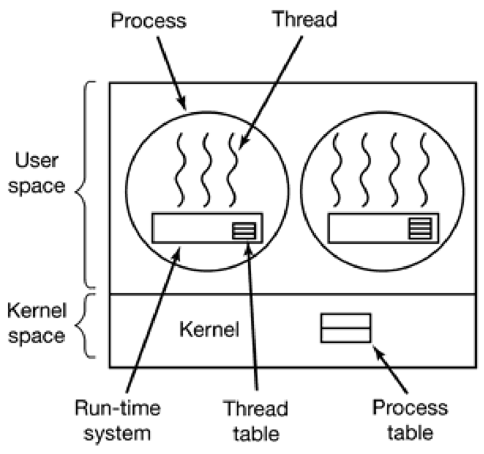


Figure 2.11: A user-level thread package.

How thread switching is performed in user-level thread package [8]

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.

Difference between process scheduling (switching) and thread scheduling

During thread switching, the procedure that saves the thread's state and the scheduler are just local procedures, so invoking them is much more efficient than making a kernel call. Among other issues, no trap is needed, no context switch is needed, the memory cache need not be flushed, and so on. This makes thread scheduling very fast.

Advantages of user-level thread package

1. As user-level thread packages are put entirely in user space and the kernel knows nothing about them, therefore, they can be implemented on an operating system that does not support threads.

[8] This is similar to normal process context switching.

2. Thread switching in user-level thread package is faster as it is done by local procedures (of the run-time system) rather than trapping the kernel.
3. User-level threads allow each process to have its own customized scheduling algorithm.
4. User-level threads scale better, since kernel threads invariably require some table space and stack space in the kernel, which can be a problem if there are a very large number of threads.

Problems of user-level thread package

1. **How blocking system calls are implemented**

   If a thread makes a blocking system call, the kernel blocks the entire process. Thus, the other threads are also blocked. But one of the main goals of having threads in the first place was to allow each one to use blocking calls, but to prevent one blocked thread from affecting the others.

   Possible solutions for this problem:

   a. The system calls could all be changed to be non-blocking (e.g., a read on the keyboard would just return 0 bytes if no characters were already buffered).

      Problems with this solution:

      i. Requiring changes to the operating system is unattractive.
      ii. One of the arguments for user-level threads was precisely that they could run with *existing* operating systems.
      iii. Changing the semantics of read will require changes to many user programs.

   b. A *wrapper* code around the system calls could be used to tell in advance if a system call will block. In some 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 only does the `read` call if it is safe (i.e., will not block). If the `read` call will block, the call is not made. Instead, another thread is run. The next time the run-time system gets control, it can check again to see if the `read` is now safe.

      Problem with this solution:

      This approach requires rewriting parts of the system call library, is inefficient and inelegant.

2. **How page faults are handled**[9]

   If a thread causes a page fault, the kernel, not even knowing about the existence of threads, naturally blocks the entire process until the disk I/O is complete, even though other threads might be runnable.

3. **Will it be possible to schedule (i.e., switch) threads?**

   If a thread starts running, no other thread in that process will ever run unless the first thread voluntarily gives up the CPU. Within a single process, there are no clock interrupts, making it impossible to schedule threads in round-robin fashion (taking turns). Unless a thread enters the run-time system (i.e., call any thread library function) of its own free will, the scheduler will never get a chance.

   Possible solution for this problem:

   The run-time system may request a clock signal (interrupt) once a second to give it control.

   Problems with this solution:

   i. This is crude and messy to program.

[9] This is somewhat analogous to the problem of blocking system calls.

ii. Periodic clock interrupts at a higher frequency (e.g., once a millisecond instead of once a second) are not always possible; and even if they are, the total overhead may be substantial.

iii. A thread might also need a clock interrupt, hence it will interfere with the run-time system's use of the clock.

4. Do we need a user-level thread package *at all*?

Programmers generally want threads precisely in applications where the threads block often, as, for example, in a multithreaded web server. These threads are constantly making system calls. Once a trap has occurred to the kernel to carry out the system call, it is hardly any more work for the kernel to switch threads if the old one has blocked, and having the kernel do this eliminates the need for constantly making `select` system calls that check to see if `read` system calls are safe. So, instead of using user-level threads, we can use kernel-level threads.

For applications that are essentially entirely CPU bound and rarely block, what is the point of having threads at all? No one would seriously propose computing the first *n* prime numbers or playing chess using threads because there is nothing to be gained by doing it that way.

Final remarks on user-level threads

User-level threads yield good performance, but they need to use a lot of tricks to make things work!

2.12 Implementing threads in the kernel (Kernel-level thread package)

Organization of threads in kernel space

- The thread table is in the kernel space. 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.
- There is no thread table in each process.
- No run-time system is needed in each process.

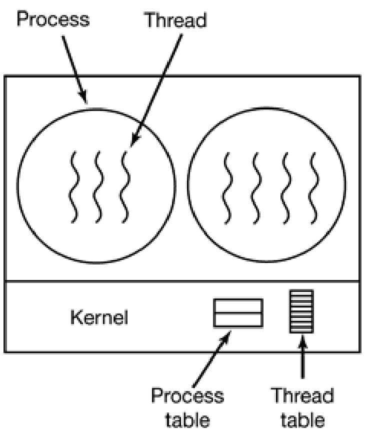


Figure 2.12: Kernel-level thread package.

How thread switching is performed in kernel-level thread package
Or, *difference in thread switching between user-level and kernel-level threads*

All calls that might block a thread are implemented as system calls, at considerably greater cost than a call to a run-time system procedure. 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).

Thread recycling: A technique to improve performance of kernel-level threads

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.

Advantages of kernel-level threads

- Do not require any new, non-blocking system calls.
- 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.

Disadvantage of kernel-level threads

The cost of a system call is substantial, so if thread operations (creation, termination, etc.) are used frequently, much more overhead will be incurred.

2.13 Hybrid implementation of threads and schedular activations

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 the kernel threads.

In this design, 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.

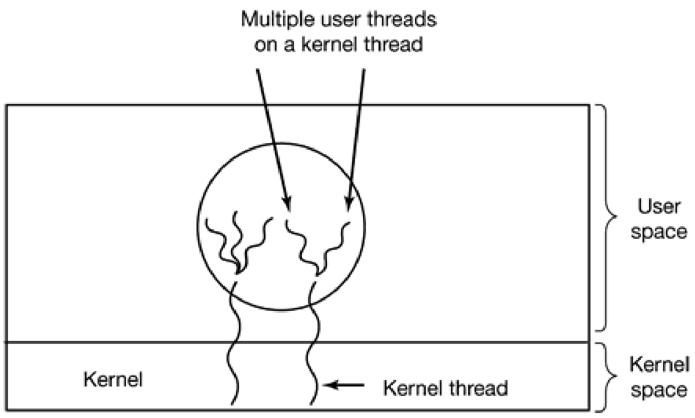


Figure 2.13(a): Multiplexing user-level threads onto kernel-level threads.

Schedular activations: an approach towards the hybrid implementation of threads

Goals of schedular activations

- To combine the advantage of user threads (good performance) with the advantage of kernel threads (not having to use a lot of tricks to make things work).
- User threads should not have to make special non-blocking 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. The user-space run-time system can block the synchronizing thread and schedule a new one by itself.

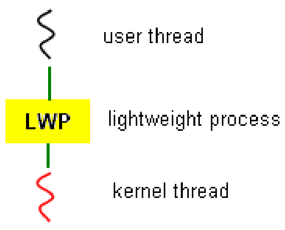


Figure 2.13(b): LWP.

Implementing schedular activations

Lightweight processes: the communication medium between kernel threads and user threads

- To connect user-level threads with kernel-level threads, an intermediate data structure — typically known as a *lightweight process*, or LWP, shown in *figure 2.13(b)* — between the user and kernel thread is placed.
- To the user-thread library, the LWP appears to be a *virtual processor* on which the application can schedule a user thread to run.
- Each LWP is attached to a kernel thread, and it is the kernel threads that the operating system schedules to run on physical processors.
- If a kernel thread blocks (such as while waiting for an I/O operation to complete), the LWP blocks as well. Up the chain, the user-level thread attached to the LWP also blocks.

Allocation of lightweight processes to user-level threads

- The kernel assigns a certain number of virtual processors to each process and lets the (user-space) run-time system allocate threads to those virtual processors.
- The number of virtual processors allocated to a process is initially one, but the process can ask

for more and can also return processors it no longer needs.

- The kernel can also take back virtual processors already allocated in order to assign them to other, more needy, processes.

How a thread block (e.g. executing a blocking system call or causing a page fault) is handled

- 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, roughly analogous to a signal in UNIX. This mechanism is called an *upcall*.
- The kernel then allocates a new virtual processor to the application. The application runs an upcall handler on this new virtual processor, which saves the state of the blocking thread and relinquishes the virtual processor on which the blocking thread is running.
- The upcall handler then schedules another thread that is eligible to run on the new virtual processor.
- Later, when the event that the blocking thread was waiting for occurs, the kernel makes another upcall to the thread library informing it that the previously blocked thread is now eligible to run. The upcall handler for this event also requires a virtual processor, and the kernel may allocate a new virtual processor or preempt one of the user threads and run the upcall handler on its virtual processor.
- After marking the unblocked thread as eligible to run, it is then up to the run-time system to decide which thread to schedule on that CPU: the preempted one, the newly ready one, or some third choice.

Objection to schedular activations

An objection to scheduler activations is the fundamental reliance on upcalls, a concept that violates the structure inherent in any layered system. Normally, layer n offers certain services that layer $n + 1$ can call on, but layer n may not call procedures in layer $n + 1$. Upcalls do not follow this fundamental principle.

2.14 IPC (Interprocess Communication)

Processes frequently need to communicate with other processes. For example, in a shell pipeline, the output of the first process must be passed to the second process, and so on down the line. Thus there is a need for communication between processes.

Issues related to IPC

Very briefly, there are three issues here:

1. How one process can pass information to another.
2. How we can make sure two or more processes do not get into each other's way when engaging in critical activities (suppose two processes each try to grab the last 1 MB of memory).
3. How we can ensure proper sequencing when dependencies are present: if process *A* produces data and process *B* prints them, *B* has to wait until *A* has produced some data before starting to print.

How we can pass information between processes

We have two ways of passing information between processes:

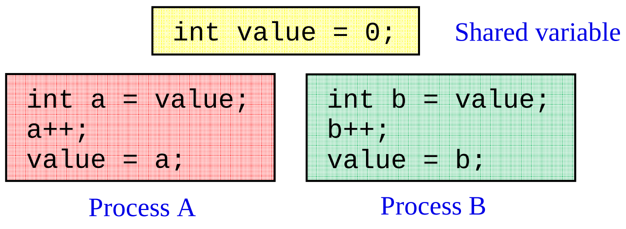
1. Shared-memory technique
2. Message-passing technique

How we can ensure synchronization

For this, we first need to understand when synchronization problem occurs.

2.15 Race Conditions

Consider the situation in *figure 2.15*. We have a shared variable named `value`. There are two processes – A and B, both of whom try to increase the value of `value`. Now, Process A increments `value` only one time and the same goes for Process B. So, after both the processes finish their tasks, `value` should contain 2.



```
int value = 0;
```

```
int a = value;
a++;
value = a;
```

```
int b = value;
b++;
value = b;
```

Figure 2.15: Race Condition.

Suppose, first, Process A executes its first instruction and stores the value of `value` (i.e., 0) in `a`. Right after it, a context switch occurs and Process B completes its task updating the value of `value` as 1. Now, Process A resumes and completes its task – incrementing `a` (thus yielding 1) and updating `value` with the value of `a` (i.e., 1). Therefore, after both the processes finish their tasks, `value` now contains 1 instead of 2.

Situations like this, where two or more processes are reading or writing some shared data and the final result depends on who runs precisely when, are called *race conditions*. Debugging programs containing race conditions is no fun at all. The results of most test runs are fine, but once in a rare while something weird and unexplained happens.

Critical Region / Critical Section

That part of the program where the shared resource is accessed is called the *critical region* or *critical section*.

Conditions that must be met to solve critical-section problem

If we could arrange matters such that no two processes were ever in their critical regions at the same time, we could avoid races. Although this requirement avoids race conditions, this is not sufficient for having parallel processes cooperate correctly and efficiently using shared data. We need four conditions to hold to have a good solution:

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

2.16 Approach towards solving race condition situations

First, let's try to find out *why* race condition occurs. If some process is in the middle of its critical section while the context switch occurs, then race condition occurs. So, our approach should be any of the followings:

1. Either temporarily disable context switching when one process is inside its critical region.
2. Use some techniques so that even if context switch occurs in the middle of executing critical section, no other processes could enter their critical sections. [Typically *mutual exclusion*]

2.17 Disabling interrupts

Each process would 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, after all, and with interrupts turned off the CPU will not be switched to another process. Thus, once a process has disabled interrupts, it can examine and update the shared memory without fear that any other process will intervene.

Problems with disabling interrupts

1. It is unwise to give user processes the power to turn off interrupts. Suppose that one of them did it and never turned them on again? That could be the end of the system.
2. 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.

3. It is frequently convenient for the kernel itself to disable interrupts for a few instructions while it is updating variables or lists. If an interrupt occurred while the list of ready processes, for example, was in an inconsistent state, race conditions could occur.

Comments on this technique

Disabling interrupts is often a useful technique within the operating system itself but is not appropriate as a general mutual exclusion mechanism for user processes.

2.18 Using some other techniques

The key to preventing trouble here is to find some way to prohibit more than one process from reading and writing the shared data at the same time. Put in other words, what we need is *mutual exclusion*, that is, 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 difficulty above occurred because process *B* started using one of the shared variables before process *A* was finished with it.

There are two ways of implementing mutual exclusion in *shared-memory systems*:

1. Using busy waiting
2. Using sleep and wakeup

2.19 Mutual exclusion with busy waiting

In this section we will examine various proposals for achieving mutual exclusion, so that while one process is busy updating shared memory in its critical region, no other process will enter *its* critical region and cause trouble.

Lock Variables

A single, shared (lock) variable is used. If the value of the variable is 0, it means that no process is in its critical region; and if it is 1, it means that some process is in its critical region. Initially, the value is set to 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.

Problem with lock variables

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

A variable *turn*, initially 0, keeps track of whose turn it is to enter the critical region. 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.

```
while (TRUE) {
    while (turn != 0) {}     //loop
    critical_region();
    turn = 1;
    noncritical_region();
}
```
(a)

```
while (TRUE) {
    while (turn != 1) {}     //loop
    critical_region();
    turn = 0;
    noncritical_region();
}
```
(b)

Figure 2.15(a): A proposed solution to the critical region problem. **(a)** Process 0. **(b)** Process 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***.

Problem with strict alternation [10]

Suppose that process 1 finishes its critical region quickly, so both processes are in their non-critical regions, with *turn* set to 0. Now, process 0 executes its whole loop quickly, exiting its critical region and setting *turn* to 1. At this point *turn* is 1 and both processes are executing in their noncritical regions.

Suddenly, process 0 finishes its noncritical region and goes back to the top of its loop. Unfortunately, it is not permitted to enter its critical region now, because *turn* is 1 and process 1 is busy with its noncritical region. It hangs in its while loop until process 1 sets *turn* to 0. Put differently, taking turns is not a good idea when one of the processes is much slower than the other.

This situation violates condition 3 (of the conditions for a good solution to IPC problems): process 0 is being blocked by a process not in its critical region.

Comments on this technique

In fact, this solution requires that the two processes strictly alternate in entering their critical regions. Neither one would be permitted to enter its critical section twice in a row.

Peterson's Solution

This is a solution combining the idea of taking turns with the idea of lock variables.

- There are two functions – `enter_region()` and `leave_region()`, a `turn` variable and an `interested` array. (The `turn` variable and the `interested` array combinely acts as a *lock* variable).
- 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 executing its critical region, the process calls `leave_region()` to indicate that it is done and to allow the other process to enter, if it so desires.

```
#define FALSE 0
#define TRUE  1
#define N     2                     // number of processes

int turn;                           // whose turn is it?
int interested[N];                  // all values initially 0 (FALSE)

void enter_region(int process) {    // process is 0 or 1
    int other = 1 - process;        // process number of the other process
    interested[process] = TRUE;     // show that you are interested
    turn = process;                 // set flag
    while (turn == process && interested[other] == TRUE) {} // null statement
}

void leave_region (int process) {   // process, who is leaving
    interested[process] = FALSE;    // indicate departure from critical region
}
```

Figure 2.15(b): Peterson's solution for achieving mutual exclusion.

[10] For a better understanding, please see the slide *Problem with Strict Alternation* accompanied with this book.

How this solution works

Initially neither process is in its critical region. Now process 0 calls `enter_region()`. It indicates its interest by setting its array element and sets `turn` to 0. Since process 1 is not interested, `enter_region()` returns immediately. If process 1 now calls `enter_region()`, it will hang there until `interested[0]` goes to `FALSE`, an event that only happens when process 0 calls `leave_region()` to exit the critical region.

Peculiarity of this solution [11]

Consider the case when both processes call `enter_region()` almost simultaneously. Both will store their process number in `turn`. Whichever store is done last is the one that counts; the first one is overwritten and lost. Suppose that process 1 stores last, so `turn` is 1. When both processes come to the while statement, process 0 executes it zero times and enters its critical region. Process 1 loops and does not enter its critical region until process 0 exits its critical region.

However, this is not a problem; because, after all, no race condition is occurring.

The TSL (Test and Set Lock) Instruction

We can use a lock variable which is controlled by hardware. Many computers, especially those designed with multiple processors in mind, have an instruction:

```
TSL RX, LOCK
```

This instruction works as follows. It reads the contents of a *shared* memory word (i.e., the variable) `lock` into register `RX` and then stores a nonzero value at the memory address `lock`. The operations of reading the word and storing into it are guaranteed to be indivisible — no other processor can access the memory word until the instruction is finished. The CPU executing the TSL instruction locks the memory bus to prohibit other CPUs from accessing memory until it is done.

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. While one process is executing its critical region, another process tests the value of lock, and if it is 1, it continues checking the value untile the value becomes 0 (which happens only when the former process exits its critical region).

```
enter_region:
    TSL REGISTER, LOCK    ; copy lock to register and set lock to 1
    CMP REGISTER, 0       ; was lock zero?
    JNE enter_region      ; if it was non-zero, lock was set, so loop
    RET                   ; return to caller. critical region entered

leave_region:
    MOV LOCK, 0           ; store a zero in lock
    RET                   ; return to caller
```

Figure 2.15(c): Entering and leaving a critical region using the TSL instruction.

Problems with the busy waiting approach

- Wastes CPU time.
- The *priority inversion problem*:

  Consider a computer with two processes, *H*, with high priority and *L*, with low priority. 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.

[11] For a better understanding, please see the slide *Peculiarity of Peterson's Solution* accompanied with this book.

General procedure for solving synchronization problem using busy waiting approach

Before entering its critical region, a process calls *enter_region*, which does busy waiting until the lock is free; then it acquires the lock and returns. After the critical region the process calls *leave_region*, which stores a 0 in the *lock*.

2.20 Mutual exclusion with `sleep` and `wakeup`

Now let us look at some interprocess communication primitives that block instead of wasting CPU time when they are not allowed to enter their critical regions. One of the simplest is the pair `sleep` and `wakeup`. `sleep` is a system call that causes the caller to block, that is, be suspended until another process wakes it up. The `wakeup` call has one parameter, the process to be awakened. Alternatively, both `sleep` and `wakeup` each have one parameter, a memory address used to match up `sleep`s with `wakeup`s.

The producer-consumer problem (a.k.a. the bounded-buffer 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).

Description and a general solution[12] of the 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.

Trouble arises when the producer wants to put a new item in the buffer, but it is already full. The solution is for the producer to go to sleep, to be awakened when the consumer has removed one or more items. Similarly, if the consumer wants to remove an item from the buffer and sees that the buffer is empty, it goes to sleep until the producer puts something in the buffer and wakes it up.

Implementing the general solution

To keep track of the number of items in the buffer, we will need a variable, `count`. If the maximum number of items the buffer can hold is `N`, the producer's code will first test to see if `count` is `N`. If it is, the producer will go to sleep; if it is not, the producer will add an item and increment `count`.

The consumer's code is similar: first test `count` to see if it is 0. If it is, go to sleep, if it is non-zero, remove an item and decrement the counter. Each of the processes also tests to see if the other should be awakened, and if so, wakes it up.

```
#define N 100                          /* number of slots in the buffer */
int count = 0;                         /* number of items in the buffer */

void producer(void) {
    while (TRUE) {                     /* repeat forever */
        int item = produce_item();     /* generate next item */
        if (count == N)                /* if buffer is full, go to sleep */
            sleep();
        insert_item(item);             /* put item in buffer */
        count = count + 1;             /* increment count of items in buffer */
        if (count == 1)                /* was buffer empty? */
            wakeup(consumer);
    }
}

void consumer(void) {
    while (TRUE) {                     /* repeat forever */
        if (count == 0)                /* if buffer is empty, got to sleep */
            sleep();
        int item = remove_item();      /* take item out of buffer */
        count = count − 1;             /* decrement count of items in buffer */
        if (count == N − 1)            /* was buffer full? */
            wakeup(producer);
        consume_item(item);            /* print item */
    }
}
```

Figure 2.16(a): The producer-consumer problem with a fatal race condition.

[12] For a better understanding of the solution, please see the slide *The Producer-Consumer Problem* accompanied with this book.

The situation when deadlock occurs

Suppose, the buffer is empty and the consumer has just read `count` to see if it is 0. At that instant, the scheduler decides to stop running the consumer temporarily and start running the producer. The producer inserts an item in the buffer, increments `count`, and notices that it is now 1. Reasoning that `count` was just 0, and thus the consumer must be sleeping, the producer calls `wakeup()` to wake the consumer up.

Unfortunately, the consumer is not yet logically asleep, so the `wakeup` signal is lost. When the consumer next runs, it will test the value of `count` it previously read, find it to be 0, and go to sleep. Sooner or later the producer will fill up the buffer and also go to sleep. Both will sleep forever.

The essence of the problem here is that a `wakeup` sent to a process that is not (yet) sleeping is lost. If it were not lost, everything would work.

A quick fix to this problem

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. The wakeup waiting bit is a piggy bank for wakeup signals.

Problem with this quick fix

While the wakeup waiting bit saves the day in this simple example, it is easy to construct examples with three or more processes in which one wakeup waiting bit is insufficient. We could make another patch and add a second wakeup waiting bit, or maybe 8 or 32 of them, but in principle the problem is still there.

Semaphores / Counting Semaphores

This was the situation in 1965, when E. W. Dijkstra (1965) suggested using an integer variable to count the number of wakeups saved for future use. In his proposal, a new variable type, 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.

- There are two operations on a semaphore – `down` and `up` (or, `sleep` and `wakeup` respectively).
- The `down` operation on a semaphore checks to see if the value is greater than 0. If so, it decrements the value (i.e., uses up one stored wakeup) and just continues. If the value is 0, the process is put to sleep without completing the `down` for the moment.
- The `up` operation increments the value of the semaphore addressed. If one or more processes were sleeping on that semaphore, unable to complete an earlier `down` operation, one of them is chosen by the system (e.g., at random) and is allowed to complete its `down`. Thus, after an `up` on a semaphore with processes sleeping on it, the semaphore will still be 0, but there will be *one* fewer process sleeping on it.
- It is guaranteed that once a semaphore operation has started, no other process can access the semaphore until the operation has completed or blocked.

Various terminologies for the semaphore operations

Dijkstra	**`P()` [*proberen* ("to test")]**	**`V()` [*verhogen* ("to increment")]**
Tanenbaum	`down()`	`up()`
Silberschatz	`wait()`	`signal()`
POSIX	`sem_wait()`	`sem_post()`

Operation	`while (semaphore == 0) {`
	`        sleep();`
	`}`
	`semaphore--;`
		`semaphore++;`
		`if (blocked_processes_exist) {`
		`        wakeup_any_of_them();`
		`}`

Solution to the producer-consumer problem using semaphores

- We need to maintain two constraints:
  1. ***Proper sequencing for maintaining dependency (i.e., synchronization)***: When the buffer is full, the producer would wait until the consumer takes out data from the buffer; and when it is empty, the consumer would wait until the producer puts some data into the buffer.
  2. ***Avoiding race condition***: So that both the producer and the consumer do not access the buffer at the same time.
- For maintaining the dependency, we use two semaphores – `empty` (initialized to N – the number of slots in the buffer) and `full` (intialized to 0). `empty` would count the number of empty buffer slots, whereas `full` would count the number of filled buffer slots.

  A question might arise, why two semaphores instead of one (e.g., `count`)? The answer: that's because the producer needs to block when the buffer is full (i.e., `count = N`); and according to the characteristics of semaphores, a process can be blocked only if the value of a semaphore is equal to *zero*.

  So, the producer would check `empty` and block if it is zero, and the consumer would check `full` and block if it is zero.
- To avoid race condition, we use a semaphore named `mutex` (initialized to 1). This semaphore acts like a *lock* variable. A process would `down mutex` to *zero* and then enter its critical section. When it is finished with the critical section, it would `up mutex` to 1. When `down`ing `mutex` to zero, if a process finds it to be already zero, then it blocks.

```c
#define N 100                      /* number of slots in the buffer */
typedef int semaphore;             /* semaphores are a special kind of int */
semaphore mutex = 1;               /* controls access to critical region */
semaphore empty = N;               /* counts empty buffer slots */
semaphore full = 0;                /* counts full buffer slots */

void producer(void) {
    while (TRUE) {                 /* TRUE is the constant 1 */
        int item = produce_item(); /* generate something to put in buffer */

        down(&empty);              /* decrement empty count */
        down(&mutex);              /* enter critical region */

        insert_item(item);         /* put new item in buffer */

        up(&mutex);                /* leave critical region */
        up(&full);                 /* increment count of full slots */
    }
}

void consumer(void) {
    while (TRUE) {                 /* infinite loop */
        down(&full);               /* decrement full count */
        down(&mutex);              /* enter critical region */

        int item = remove_item(); /* take item from buffer */

        up(&mutex);                /* leave critical region */
        up(&empty);                /* increment count of empty slots */

        consume_item(item);        /* do something with the item */
    }
}
```

Mutexes[13] / Locks / Mutex Locks / Binary Semaphores[14]

- A *mutex* is a variable that can be in one of two states: unlocked (0) or locked (1).
- It is a simplified version of the semaphore, and is used when the semaphore's ability to count is not needed. Mutexes are good only for managing mutual exclusion to some shared resource or piece of code.
- However, mutexes can also be used to *signal* completion of code across threads.
- They are easy and efficient to implement, which makes them especially useful in user-level thread packages.
- Mutexes have ownership, unlike semaphores. Although any thread, within the scope of a mutex, can get an unlocked mutex and lock access to the same critical section of code, only the thread that locked a mutex can unlock it.
- Mutexes can be implemented via semaphores or the TSL instruction.

Implementing mutexes via semaphores

See *Solution to the producer-consumer problem using semaphores* above.

Implementing mutexes (in user-space) via TSL instruction

Below are two codes using TSL instruction – one is the implementation of busy waiting (which we have seen before), and the other is the implementation of mutexes. Both are given to differentiate between the two.

Busy-waiting	Blocking
`enter_region:`	`mutex_lock:`
`    TSL REGISTER, LOCK`	`    TSL REGISTER, MUTEX`
`    CMP REGISTER, 0`	`    CMP REGISTER, 0       ;was mutex zero?`
`    JNE enter_region`	`    JZE ok         ;if it was zero, mutex was unlocked, so return`
`    RET`	`    CALL thread_yield    ;mutex is busy; schedule another thread`
	`    JMP mutex_lock        ;try again later`
	`ok:`
	`    RET     ;return to caller, critical region entered`
`leave_region:`	`mutex_unlock:`
`    MOV LOCK, 0`	`    MOV MUTEX, 0          ;store a zero in mutex`
`    RET`	`    RET                   ;return to caller`

Difference between enter_region and mutex_lock

When *enter_region* fails to enter the critical region, it keeps testing the lock repeatedly (busy waiting). Eventually, the clock runs out and some other process is scheduled to run.

On the other hand, when the *mutex_lock* fails to acquire a lock, it calls *thread_yield* to give up the CPU to another thread. Consequently there is no busy waiting. When the thread runs the next time, it tests the lock again.

Advantage of mutexes when implemented in user-level thread packages

Since *thread_yield* is just a call to the thread scheduler in user space, it is very fast. As a consequence, neither *mutex_lock* nor *mutex_unlock* requires any kernel calls. Using them, user-level threads can synchronize entirely in user space using procedures that require only a handful of instructions.

[13] Mutex is a mnemonic for MUTual EXclusion.

[14] A mutex is called a binary semaphore *only* when it is implemented using a semaphore, not in any other way (e.g. using TSL instruction.

Monitors

Disadvantage of Semaphores

The programmer must be very careful when using semaphores. One subtle error and everything comes to a grinding halt. It is like programming in assembly language, only worse, because the errors are race conditions, deadlocks, and other forms of unpredictable and irreproducible behavior.

For example, look closely at the order of the `downs` before inserting or removing items from the buffer in Fig. 2-24. Suppose that the two `downs` in the producer's code were reversed in order, so `mutex` was decremented before `empty` instead of after it. If the buffer were completely full, the producer would block, with `mutex` set to 0. Consequently, the next time the consumer tried to access the buffer, it would do a `down` on `mutex`, now 0, and block too. Both processes would stay blocked forever and no more work would ever be done. This unfortunate situation is called a *deadlock*.

To make it easier to write correct programs, Hoare (1974) and Brinch Hansen (1975) proposed a higher-level synchronization primitive called a *monitor*.

What is a monitor

- 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. Monitors are a programming language construct, so the compiler knows they are special and can handle calls to monitor procedures differently from other procedure calls. Typically, when a process calls a monitor procedure, the first few instructions of the procedure will check to see, if any other process is currently active within the monitor. If so, the calling process will be suspended until the other process has left the monitor. If no other process is using the monitor, the calling process may enter.

Let's try to understand the concept of monitor with an example of a class in Java language. Also, let's try to solve the producer-consumer problem along with. Suppose, we have two classes – Producer and Consumer. They each have an operation (i.e., method) which needs to be synchronized. Let's assume the methods are `insert()` and `remove()`. Again, both classes will try to access a shared buffer. Let's name the buffer variable as `buffer`. So, we have two methods and a variable which could cause synchronization problems.

Now, let's declare a class (named `Monitor`) and insert those two methods and the variable into it. The pseudocode for the three classes might be as follows:

```
class Producer {
    void produce() {
        while(true) {
            int item = produce_item();
            insert(item);
        }
    }
}

class Consumer {
    void consume() {
        while(true) {
            int item = remove_item();
            process(item);
        }
    }
}
```

```
class Monitor {
    int[] buffer = new int[100];
    int count = 0;

    void insert(int item) {
        buffer[count] = item;
        count++;
    }

    int remove() {
        int item = buffer[count];
        count--;
        return item;
    }
}
```

- In the above example, the `Monitor` class is the monitor package or module.
- The method calls in this class are specially handled by the compiler so that no two processes are inside the monitor at the same time.
- It is up to the compiler to implement the mutual exclusion on monitor entries, but a common way is to use a mutex or binary semaphore.
- Because the compiler, not the programmer, is arranging for the mutual exclusion, it is much less likely that something will go wrong. In any event, the person writing the monitor does not have to be aware of how the compiler arranges for mutual exclusion.

Condition variables and operations on them

It's fine that mutual exclusion has been guaranteed by using monitors. So, when one process accesses the buffer, no other process can access it. But what about the dependency synchronization? How would we ensure that the producer blocks whenever the buffer is *full* and the consumer blocks whenever the buffer is *empty*?

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 its sleeping partner by doing a `signal` on the condition variable that its partner is waiting on. The pseudocode of the `Monitor` class with the introduction of condition variables and operations on them is as follows:

```
class Monitor {
    int[] buffer = new int[100];
    int count = 0;
    condition full, empty;

    void insert(int item) {
        if (count == 100)
            wait(full);
        buffer[count] = item;
        count++;
        if (count == 1)
            signal(empty);
    }

    int remove() {
        if (count == 0)
            wait(empty);
        int item = buffer[count];
        count--;
        if (count == 100)
            signal(full);
        return item;
    }
}
```

It must be noted that condition variables are not counters or semaphores. They do not accumulate signals for later use the way semaphores do. Thus, if a condition variable is signaled with no one waiting on it, the signal is lost forever.

What happens after a `signal`

When a process `signals` the other blocked process from inside the monitor, there will be two processes inside the monitor – the newly awakened one and the one which was already inside the monitor. But this situation must not occur. So, to avoid having two active processes in the monitor at the same time, we need a rule telling what happens after a `signal`. There are three possible solutions:

1. Hoare proposed letting the newly awakened process run, suspending the other one.
2. Brinch Hansen proposed that a process doing a `signal` *must* exit the monitor immediately. In other words, a `signal` statement may appear only as the final statement in a monitor procedure.
3. There is also a third solution, not proposed by either Hoare or Brinch Hansen. This is to let the signaler continue to run and allow the waiting process to start running only after the signaler has exited the monitor.

We will use Brinch Hansen's proposal because it is conceptually simpler and is also easier to implement. 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.

Drawback of monitors

Monitors are a programming language concept. The compiler must recognize them and arrange for the mutual exclusion somehow. C, Pascal, and most other languages do not have monitors, so it is unreasonable to expect their compilers to enforce any mutual exclusion rules. In fact, how could the compiler even know which procedures were in monitors and which were not?

If it is argued that these same languages do not have semaphores either, then it can be said that adding semaphores is easy: all you need to do is add two short assembly code routines to the library to issue the `up` and `down` system calls. The compilers do not even have to know that they exist. Of course, the operating systems have to know about the semaphores, but at least if you have a semaphore-based operating system, you can still write the user programs for it in C or C++ (or even assembly language if you are masochistic enough!). With monitors, you need a language that has them built in.

2.21 How multiple processes share common memory locations

If processes have disjoint address spaces, how can they share the *turn* variable in Peterson's algorithm, or semaphores or a common buffer?

There are various ways:

1. First, some of the shared data structures, such as the semaphores, can be stored in the kernel and only accessed via system calls. This approach eliminates the problem.
2. Second, most modern operating systems (including UNIX and Windows) offer a way for processes to share some portion of their address space with other processes. In this way, buffers and other data structures can be shared.
3. In the worst case, that nothing else is possible, a shared file can be used.

If two or more processes share most or all of their address spaces, the distinction between processes and threads becomes somewhat blurred but is nevertheless present. Two processes that share a common address space still have different open files, alarm timers, and other per-process properties, whereas the threads within a single process share them. And it is always true that multiple processes sharing a common address space never have the efficiency of user-level threads since the kernel is deeply involved in their management.

2.22 Problem with shared-memory technique

A problem with monitors, and also with semaphores, is that they were designed for solving the mutual exclusion problem on one or more CPUs that all have access to a *common* memory. When we go to a distributed system consisting of multiple CPUs, each with its *own private* memory, connected by a local area network, these primitives become inapplicable.

2.23 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);`

The former call sends a message to a given destination and the latter one receives a message from a given source (or from *any*, if the receiver does not care). If no message is available, the receiver can block until one arrives. Alternatively, it can return immediately with an error code.

Design issues for message passing

1. ***Reliability*:** When message passing is used across a network, messages can be lost.

   A message receive-acknowledgement system can be used to solve this issue.

2. ***Addressing*:** How processes are named, so that the process specified in a send or receive call is unambiguous?

   Port addresses can be used to solve this.

3. ***Authentication*:** how can the client tell that he is communicating with the real file server, and not with a fake one?

4. ***Performance*:** Copying messages from one process to another *within the same machine* is always slower than doing a semaphore operation or entering a monitor.

   Much work has gone into making message passing efficient. Cheriton (1984), for example, suggested limiting message size to what will fit in the machine's registers, and then doing message passing using the registers.

5. ***Buffering:*** How sent messages should be queued?

   - **Zero capacity:** There should be no buffer. In this case, the sender must block until the recipient receives the message.
   - **Bounded capacity:** The queue has finite length *n*; thus, at most *n* messages can reside in it. If the queue is full, the sender must block until space is available in the queue.
   - **Unbounded capacity:** The queue's length is potentially infinite; thus, any number of messages can wait in it. The sender never blocks.

6. ***Synchronization*:** How the `send()` and `receive()` primitives would synchronize?

   Message passing may be either *blocking* or *nonblocking* (a.k.a. *synchronous* or *asynchronous*):

   - **Blocking send:** The sending process is blocked until the message is received by the receiving process.
   - **Nonblocking send:** The sending process sends the message and resumes operation.
   - **Blocking receive:** The receiver blocks until a message is available.
   - **Nonblocking receive:** The receiver retrieves either a valid message or a null.

2.24 Scheduling

When a computer is multiprogrammed, it frequently has multiple processes competing for the CPU at the same time. 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*.

Process Behavior [*i.e., which processes should be given more priorities*]

Nearly all processes alternate bursts of computing with (disk) I/O requests.

Some processes spend most of their time computing, while others spend most of their time waiting for I/O. The former are called *CPU-bound* or *compute-bound*; the latter are called *I/O-bound*. Compute-bound processes typically have long CPU bursts and thus infrequent I/O waits, whereas I/O-bound processes have short CPU bursts and thus frequent I/O waits.

So, if an I/O-bound process wants to run, it should get a chance quickly so that it can issue its disk request and keep the disk busy.

When to schedule

1. When a new process is created.
2. When a process exits.
3. When a process switches from the *running* state to the *waiting* state (e.g., when it blocks on I/O, a semaphore, or by executing a blocking system call).
4. When a process switches from the *running* state to the *ready* state (e.g., when an interrupt occurs).
5. When a process switches from the *waiting* state to the *ready* state (e.g., at completion of I/O).

Categories of scheduling algorithms

Scheduling algorithms can be divided into two categories with respect to how they deal with clock interrupts:

1. **Non-preemptive scheduling algorithm**: This 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 until it voluntarily releases the CPU. Even if it runs for hours, it will not be forcibly suspended. In effect, no scheduling decisions are made during clock interrupts. After clock interrupt processing has been completed, the process that was running before the interrupt is always resumed.
2. **Preemptive scheduling algorithm:** This 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.

Scheduling Algorithm Goals

1. All Systems

a. **Fairness** - giving each process a fair share of the CPU.

b. **Policy enforcement** - seeing that stated policy is carried out.

For example, if the local policy is that safety control processes get to run whenever they want to, even if it means the payroll is 30 sec late, the scheduler has to make sure this policy is enforced.

c. **Balance** - keeping all parts of the system busy.

Having some CPU-bound processes and some I/O-bound processes in memory together is a better idea than first loading and running all the CPU-bound jobs and then when they are finished, loading and running all the I/O-bound jobs. If the latter strategy is used, when the CPU-bound processes are running, they will fight for the CPU and the disk will be idle. Later, when the I/O-bound jobs come in, they will fight for the disk and the CPU will be idle. Better to keep the whole system running at once by a careful mix of processes.

2. **Batch Systems**

   a. **Throughput** - maximize jobs per hour.

   Throughput is the number of jobs per unit time that the system completes.

   b. **Turnaround time** - minimize time between submission and termination.

   Turnaround time is the statistically average time from the moment that a batch job is submitted until the moment it is completed. It measures how long the average user has to wait for the output.

   c. **CPU utilization** - keep the CPU busy all the time.

3. **Interactive Systems**

   a. **Response time** - respond to requests quickly.

   Response time is the time between issuing a command and getting the result.

   b. **Proportionality** - meet users' expectations.

   Users have an inherent (but often incorrect) idea of how long things should take. When a request that is perceived as complex takes a long time, users accept that, but when a request that is perceived as simple takes a long time, users get irritated.

4. **Real-time Systems**

   a. **Meeting deadlines** - avoid losing data.

   For example, if a computer is controlling a device that produces data at a regular rate, failure to run the data-collection process on time may result in lost data.

   b. **Predictability** - avoid quality degradation in multimedia systems.

   For example, if the audio process runs too erratically, the sound would become peculiar to hear.

2.25 Scheduling Algorithms

First-Come First-Served (FCFS)
[*Non-preemptive*] [*Applies to: Batch Systems*]

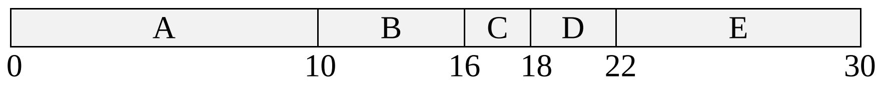
- Processes are assigned the CPU in the order they request it.
- There is a single queue of *ready* processes. When the running process blocks, the first process on the queue is run next. When a blocked process becomes ready, like a newly arrived job, it is put on the end of the queue.
- **Advantage:** Wonderful for long processes when they are executed at last.
- **Disadvantage:** Terrible for short processes if they are behind a long process.

Example

Assume the following processes are to be executed with one processor, with the processes arriving *almost* at the same time and having the following CPU burst times and priorities:

Process	CPU Burst Time (*ms*)
A	10
B	6
C	2
D	4
E	8

Gantt chart[15]**:**



Average waiting time[16]**:** (0 + 10 + 16 + 18 + 22) / 5 = **13.2 ms**

Average turnaround time: (10 + 16 + 18 + 22 + 30) / 5 = **19.2 ms**

Throughput: (5 / 30) = **0.167 processes/ms**

Shortest Job First (SJF)

[*Non-preemptive*] [*Applies to: Batch Systems*]

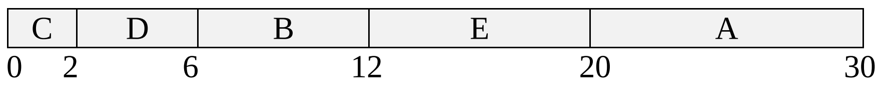
- Assumes run times are known (i.e., estimated) in advance.
- When several equally important jobs are sitting in the input queue waiting to be started, the scheduler picks the shortest job first.
- If more than one processes have the shortest running time, then FCFS is used to choose the process to be run.
- **Advantage:** Minimizes average turnaround time (if, and only if, all jobs are available at the beginning).
- **Disadvantages:**

  1. Long jobs may starve
  2. Needs to predict job length

Example

Process	CPU Burst Time (*ms*)
A	10
B	6
C	2
D	4
E	8

Gantt chart:



Average waiting time: (0 + 2 + 6 + 12 + 20) / 5 = **8 ms**

Average turnaround time: (2 + 6 + 12 + 20 + 30) / 5 = **14 ms**

Throughput: (5 / 30) = **0.167 processes/ms**

Shortest Remaining Time Next (SRTN) / Shortest Remaining Time First (SRTF) / Shortest Time to Completion First (STCF) / Preemptive Shortest Job First (Preemptive SJF)

[Preemptive] [*Applies to: Batch Systems*]

- Assumes run times are known (i.e., estimated) in advance.
- The scheduler always chooses the process whose *remaining* run time is the shortest.

[15] **Gantt chart:** A type of bar chart that illustrates a project schedule.

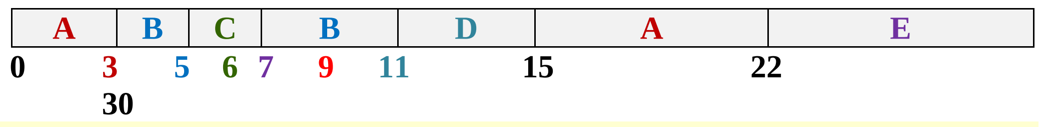
[16] **Waiting time:** Amount of time spent by a process which is ready to run but not running. In other words, it is the sum of the periods spent waiting in the ready queue.

- When a new job arrives, its total time is compared to the current process' remaining time. If the new job needs less time to finish than the current process, the current process is suspended (and placed at the end of the queue) and the new job is started.

Example

Process	Arrival Time	CPU Burst Time (*ms*)
A	0	10
B	3	6
C	5	2
D	6	4
E	9	8

Gantt chart:



A (7)	B (4)	C (1)	B (4)	B (2)	D (4)	A (7)	E (8)	[Process (Remaining time)]
B (6)	A (7)	A (7)	D (4)	D (4)	A (7)	E (8)		
	C (2)	B (4)	A (7)	A (7)	E (8)			
		D (4)		E (8)				Queue when new processes arrive

A (7)	A (7)	B (4)	D (4)	D (4)	A (7)	E (8)	
	B (4)	D (4)	A (7)	A (7)	E (8)		
		A (7)		E (8)			Queue after scheduling

Average waiting time:

(((15 – 3) + (0 - 0)) + ((7 – 5) + (3 – 3)) + (5 - 5) + (11 - 6) + (22 - 9)) / 5 = **6.4 ms**

Average turnaround time: ((22 – 0) + (11 – 3) + (7 - 5) + (15 - 6) + (30 - 9)) / 5 = **12.4 ms**

Throughput: (5 / 30) = **0.167 processes/ms**

Round-Robin Scheduling (RR Scheduling)

[*Preemptive*] [*Applies to: Both Batch and Interactive Systems*]

- Each process is assigned a time interval, called a ***time quantum*** or ***time slice***, in 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.
- When the process uses up its quantum, it is put on the *end* of the list.
- Setting the quantum too short causes too many process switches and lowers the CPU efficiency, but setting it too long may cause poor response to short interactive requests.

  For example, suppose the quantum is set to 4 ms. If the process switch overhead is 1 ms, then 20% of the CPU time will be wasted on administrative overhead.

  Again, suppose the quantum is set to 100 ms. Now, the wasted time is only 1 percent. But consider what happens on a timesharing system if ten interactive users hit the carriage return key at roughly the same time. Ten processes will be put on the list of runnable processes. If the CPU is idle, the first one will start immediately, the second one may not start until 100 msec later, and so on. The unlucky last one may have to wait 1 sec before getting a chance, assuming all the others use their full quanta. Most users will perceive a 1-sec response to a short command as sluggish.

- **Advantage:** Fair; easy to implement.
- **Disadvantage:** Assumes everybody is equal.

Example

Process	CPU Burst Time (*ms*)
A	10
B	6
C	2
D	4
E	8

Time Quantum* = 3 *ms

Gantt chart:

```
|  A  |  B  | C |  D  |  E  |  A  |  B  | D |  E  |  A  | E | A |
0     3     6   8     11    14    17    20  21    24    27  29  30
```

Average waiting time:

(((0 – 0) + (14 – 3) + (24 – 17) + (29 – 27)) + ((3 – 0) + (17 – 6)) + (6 – 0) + ((8 – 0) + (20 – 11)) + ((11 – 0) + (21 – 14) + (27 – 24))) / 5 = **15.6 ms**

Average turnaround time: (30 + 20 + 8 + 21 + 29) / 5 = **21.6 ms**

Throughput: (5 / 30) = **0.167 processes/ms**

Priority Scheduling

[*Preemptive / Non-preemptive*] [*Applies to: Both Batch and Interactive Systems*]

- Each process is assigned a priority, and the runnable process with the highest priority is allowed to run.
- Priority scheduling can be either *preemptive* or *nonpreemptive*.

  A preemptive priority scheduling algorithm will preempt the CPU if the priority of the newly arrived process is higher than the priority of the currently running process.

  A nonpreemptive priority scheduling algorithm will simply put the new process at the head of the ready queue.
- A major problem with priority scheduling algorithms is *indefinite blocking*, or *starvation*. To prevent high-priority processes from running indefinitely and low-priority processes from starving, a scheduler can do any of the following:
  - ***Aging technique***: Decrease the priority of the currently running process at each clock tick (i.e., at each clock interrupt). If this action causes its priority to drop below that of the next highest process, a process switch occurs.
  - Each process may be assigned a maximum time quantum that it is allowed to run. When this quantum is used up, the next highest priority process is given a chance to run.
- ***How priorities can be assigned to processes:***
  - **Statically / Externally**

    For example, on a military computer, processes started by generals might begin at priority 100, processes started by colonels at 90, majors at 80, captains at 70, lieutenants at 60, and so on.
  - **Dynamically / Internally**

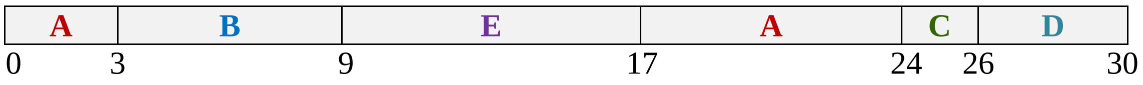
    For example, some processes are highly I/O bound and spend most of their time waiting for I/O to complete. Whenever such a process wants the CPU, it should be given the CPU immediately, to let it start its next I/O request which can then proceed in parallel with another process actually computing. Making the I/O bound process wait a long time for the CPU will just mean having it around occupying memory for an unnecessarily long time.

A simple algorithm for giving good service to I/O bound processes is to set the priority, to *1 / f*, where *f* is the fraction of the last quantum that a process used. A process that used only 1 msec of its 50 msec quantum would get priority 50, while a process that ran 25 msec before blocking would get priority 2, and a process that used the whole quantum would get priority 1.

Example (Preemptive)

Process	Arrival Time	CPU Burst Time (*ms*)	Priority
A	0	10	3
B	3	6	5 (*highest*)
C	5	2	2
D	6	4	1
E	9	8	4

Gantt chart:



Average waiting time:

(((0 – 0) + (17 – 3)) + (3 – 3) + (24 – 5) + (26 – 6) + (9 – 9)) / 5 = **10.6 ms**

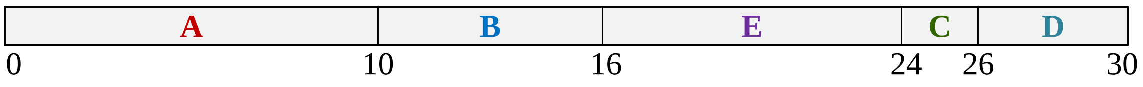
Average turnaround time: ((24 – 0) + (9 – 3) + (26 – 5) + (30 – 6) + (17 – 9)) / 5 = **16.6 ms**

Throughput: (5 / 30) = **0.167 processes/ms**

Example (Non-Preemptive)

Process	Arrival Time	CPU Burst Time (*ms*)	Priority
A	0	10	3
B	3	6	5 (*highest*)
C	5	2	2
D	6	4	1
E	9	8	4

Gantt chart:



Average waiting time: ((0 – 0) + (10 – 3) + (24 – 5) + (26 – 6) + (16 – 9)) / 5 = **10.6 ms**

Average turnaround time: ((10 – 0) + (16 – 3) + (26 – 5) + (30 – 6) + (24 – 9)) / 5 = **16.6 ms**

Throughput: (5 / 30) = **0.167 processes/ms**

Multiple / Multilevel Queue Scheduling (MQS / MQ Scheduling)
[*Fixed-Priority Preemptive (Commonly)*] [*Applies to: All the systems*]

Another class of scheduling algorithms has been created for situations in which processes are easily classified into different groups. For example, a common division is made between *foreground* (interactive) processes and *background* (batch) processes. These two types of processes have different response-time requirements and so may have different scheduling needs. In addition, foreground processes may have priority (externally defined) over background processes.

- A multilevel queue scheduling algorithm partitions the ready queue into several separate queues [*figure 2.25(a)*]. The processes are permanently assigned to one queue, generally based on some property of the process, such as memory size, process priority, or process type.
- Each queue has its own scheduling algorithm. For example, separate queues might be used for foreground and background processes. The foreground queue might be scheduled by an RR algorithm, while the background queue is scheduled by an FCFS algorithm.

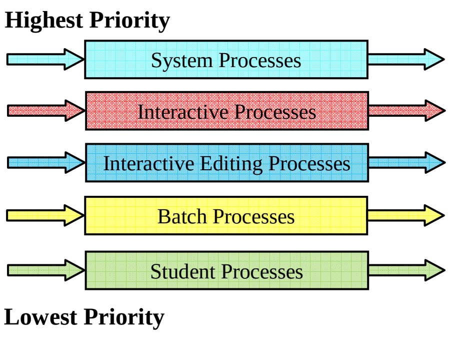


Figure 2.25(a): Multilevel queue scheduling.

- There must be scheduling among the queues, which is commonly implemented as fixed-priority preemptive scheduling. For example, the foreground queue may have absolute priority over the background queue.

Multilevel Feedback-Queue Scheduling (MFQS / MFQ Scheduling) / Exponential Queue

[*Fixed-Priority Preemptive (Commonly)*] [*Applies to: All the systems*]

Normally, in Multilevel Queue Scheduling, processes are permanently assigned to a queue and do not move from one queue to another. Therefore, lower-priority processes might starve.

The *Multilevel Feedback-Queue Scheduling* algorithm, in contrast, allows a process to move between queues. The idea is to separate processes according to the *characteristics of their CPU bursts*. If a process uses too much CPU time, it will be moved to a lower-priority queue. This scheme leaves I/O-bound and interactive processes in the higher-priority queues. In addition, a process that waits too long in a lower-priority queue may be moved to a higher-priority queue. This form of aging prevents starvation.

Example

A process entering the ready queue is put in queue 0. A process in queue 0 is given a time quantum of 8 milliseconds. If it does not finish within this time, it is moved to the tail of queue 1. If queue 0 is empty, the process at the head of queue 1 is given a quantum of 16 milliseconds. If it does not complete, it is preempted and is put into queue 2. Processes in queue 2 are run on an FCFS basis but are run only when queues 0 and 1 are empty.

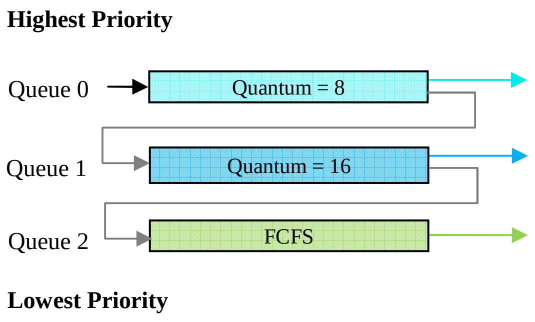


Figure 2.25(b): Multilevel feedback-queue scheduling.

This scheduling algorithm gives highest priority to any process with a CPU burst of 8 milliseconds or less. Such a process will quickly get the CPU, finish its CPU burst, and go off to its next I/O burst. Processes that need more than 8 but less than 24 milliseconds are also served quickly, although with lower priority than shorter processes. Long processes automatically sink to queue 2 and are served in FCFS order with any CPU cycles left over from queues 0 and 1.

Guaranteed Scheduling

[*Preemptive*] [*Applies to: Interactive systems*]

- Makes real promises to the users about performance and then fulfills them. One promise that is realistic to make and easy to fulfill is this: If there are *n* users logged in while you are working, you will receive about 1/*n* of the CPU power. Similarly, on a single user system with *n* processes running, all things being equal, each one should get 1/*n* of the CPU cycles.
- **Algorithm:**
  - $Ratio = \frac{\text{Actual CPU time consumed by the process since its creation}}{\text{The amount of CPU the process is entitled to}}$
  - The process with the lowest ratio is run until its ratio has moved *above* its closest competitor.

Lottery Scheduling

[*Preemptive*] [*Applies to: Interactive systems*]

- The basic idea is to give processes lottery tickets for various system resources, such as CPU time.
- Whenever a scheduling decision has to be made, a lottery ticket is chosen at random, and the process holding that ticket gets the resource.
- When applied to CPU scheduling, the system might hold a lottery 50 times a second, with each winner getting 20 ms of CPU time as a prize.
- 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 the client can run again.
- Lottery scheduling can be used to solve problems that are difficult to handle with other methods. One example is a video server in which several processes are feeding video streams to their clients, but at different frame rates. Suppose that the processes need frames at 10, 20, and 25 frames/sec. By allocating these processes 10, 20, and 25 tickets, respectively, they will automatically divide the CPU in approximately the correct proportion, that is, 10 : 20 : 25.

Fair-Share Scheduling

[*Preemptive*] [*Applies to: Interactive systems*]

So far we have assumed that each process is scheduled on its own, without regard to who its owner is. As a result, if user 1 starts up 9 processes and user 2 starts up 1 process, with round robin or equal priorities, user 1 will get 90% of the CPU and user 2 will get only 10% of it.

To prevent this situation, some systems take into account who owns a process before scheduling it. In this model, each user is allocated some fraction of the CPU and the scheduler picks processes in such a way as to enforce it. Thus 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.

As an example, consider a system with two users, each of which has been promised 50% of the CPU. User 1 has four processes, *A*, *B*, *C*, and *D*, and user 2 has only 1 process, *E*. If round-robin scheduling is used, a possible scheduling sequence that meets all the constraints is this one:

A E B E C E D E A E B E C E D E ...

On the other hand, if user 1 is entitled to twice as much CPU time as user 2, we might get

A B E C D E A B E C D E ...

Numerous other possibilities exist of course, and can be exploited, depending on what the notion of fairness is.

2.26 Scheduling in Real-Time Systems (RTS)

A *real-time* system is one in which time plays an essential role – processing must happen in a timely fashion, neither too late, nor too early.

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 events that a real-time system may have to respond to can be further categorized us ***periodic*** (occurring at regular intervals) or ***aperiodic*** (occurring unpredictably).

A system may have to respond to multiple periodic event streams. Depending on how much time each event requires for processing, it may not even be possible to handle them all. For example, if there are *m* periodic events and event *i* occurs with period P_i and requires C_i seconds of CPU time to handle each event, then the load can only be handled if

$$\sum_{i=1}^{m} \frac{C_i}{P_i} \leq 1$$

A real-time system that meets this criteria is said to be ***schedulable***.

As an example, consider a soft real-time system with three periodic events, with periods of 100, 200, and 500 msec, respectively. If these events require 50, 30, and 100 msec of CPU time per event, respectively, the system is schedulable because 0.5 + 0.15 + 0.2 < 1. If a fourth event with a period of 1 sec is added, the system will remain schedulable as long as this event does not need more than 150 msec of CPU time per event. Implicit in this calculation is the assumption that the context-switching overhead is so small that it can be ignored.

Real-time scheduling algorithms can be ***static*** or ***dynamic***. The former make their scheduling decisions before the system starts running (by assigning each process a fixed priority in advance). The latter make their scheduling decisions at run time (i.e., processes do not have fixed priorities). Static scheduling only works when there is perfect information available in advance about the work needed to be done and the deadlines that have to be met. Dynamic scheduling algorithms do not have these restrictions.

Rate Monotonic Scheduling (RMS) [*Static Scheduling Algorithm*]

It can be used for processes that meet the following conditions:

1. Each periodic process must complete within its period.
2. No process is dependent on any other process.
3. Each process needs the same amount of CPU time on each burst.
4. Any non-periodic processes have no deadlines.
5. Process preemption occurs instantaneously and with no overhead.

Example:

Let,

Process *A* runs every 30 msec, i.e., 33 times/sec; each frame requires 10 msec of CPU time.
Process *B* runs every 40 msec, i.e., 25 times/sec; each frame requires 15 msec of CPU time.
Process *C* runs every 50 msec, i.e., 20 times/sec; each frame requires 5 msec of CPU time.

∴ Priority of *A* = 33, *B* = 25 and *C* = 20.

Schedulability test: 10/30 + 15/40 + 5/50 = 0.808333 < 1
∴ The system of processes is schedulable.

In *figure* 2.26(a), initially all three processes are ready to run. The highest priority one, *A*, is chosen, and allowed to run until it completes at 10 msec, as shown in the RMS line. After it finishes, *B* and *C* are run in that order. Together, these three processes take 30 msec to run, so when *C* finishes, it is time for *A* to run again. This rotation goes on until the system goes idle at *t* = 70.

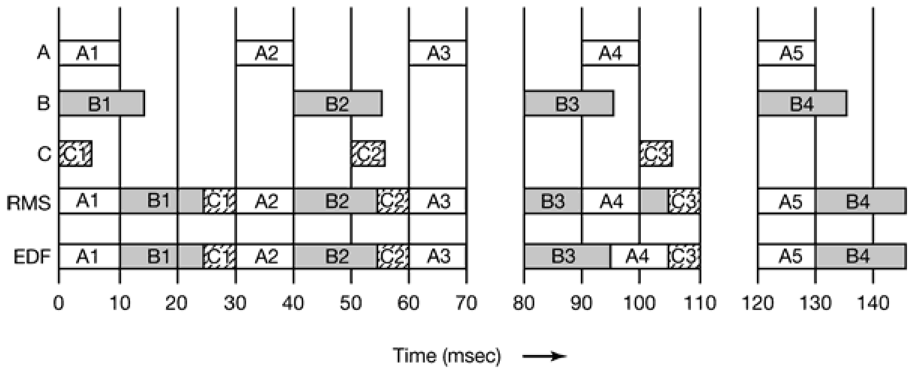


Figure 2.26(a): An example of RMS and EDF real-time scheduling.

Earliest Deadline First Scheduling (EDFS) [*Dynamic Scheduling Algorithm*]

The scheduler keeps a list of runnable processes, sorted on deadline. The algorithm runs the first process on the list, the one with the closest deadline. Whenever a new process becomes ready, the system checks to see if its deadline occurs before that of the currently running process. If so, the new process preempts the current one.

Example:

An example of EDF is given in *figure* 2.26(a). Initially all three processes are ready. They are run in the order of their deadlines, *A* must finish by $t = 30$, *B* must finish by $t = 40$, and *C* must finish by $t = 50$, so *A* has the earliest deadline and thus goes first. Up until $t = 90$ the chokes are the same as RMS. At $t = 90$, *A* becomes ready again, and its deadline is $t = 120$, the same as B's deadline. The scheduler could legitimately choose either one to run, but since in practice, preempting *B* has some nonzero cost associated with it, it is better to let *B* continue to run.

To dispel the idea that RMS and EDF always give the same results, let us now look at another example, shown in *figure* 2.26(b). In this example the periods of *A*, *B*, and *C* are the same as before, but now *A* needs 15 msec of CPU time per burst instead of only 10 msec. The schedulability test computes the CPU utilization as 0.500 + 0.375 + 0.100 = 0.975. Only 2.5% of the CPU is left over, but in theory the CPU is not oversubscribed and it should be possible to find a legal schedule.

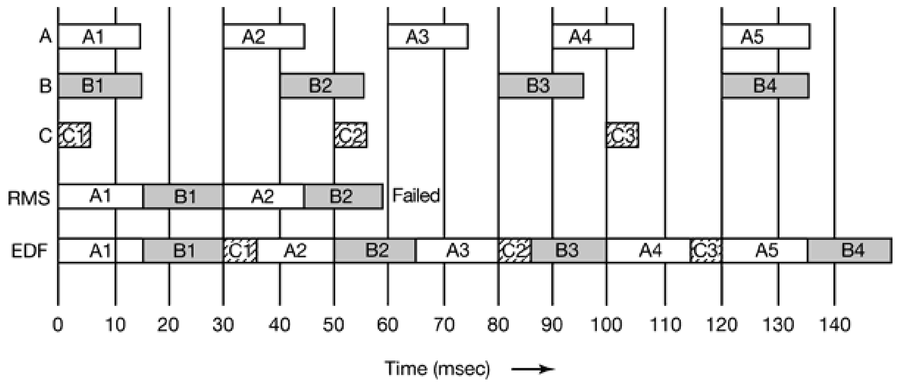


Figure 2.26(b): Example to dispel the idea that RMS and EDF always give the same results.

With RMS, the priorities of the three processes are still 33, 25, and 20 as only the period matters, not the run time. This time, *B1* does not finish until $t = 30$, at which time *A* is ready to roll again. By the time *A* is finished, at $t = 45$, *B* is ready again, so having a higher priority than *C*, it runs and *C* misses its deadline. RMS fails.

Now look at how EDF handles this case. At $t = 30$, there is a contest between *A2* and *C1*. Because *C1*'s deadline is 50 and *A2*'s deadline is 60, *C* is scheduled. This is different from RMS, where *A*'s higher priority wins.

At $t = 90$, *A* becomes ready for the fourth time. *A*'s deadline is the same as that of the current process (120), so the scheduler has a choice of preempting or not. As before, it is better not to preempt if it is not needed, so *B3* is allowed to complete.

An interesting question is why RMS failed. Basically, using static priorities only works if the CPU utilization is not too high. Liu and Layland proved that for any system of periodic processes, if

$$\sum_{i=1}^{m} \frac{C_i}{P_i} \le m\left(2^{1/m} - 1\right)$$

then RMS is guaranteed to work.

2.27 Scheduling Policy versus Scheduling Mechanism

Sometimes it happens that one process has many children running under its control. For example, a database management system process may have many children. Each child might be working on a different request, or each one might have some specific function to perform (query parsing, disk access, etc.). It is entirely possible that the main process has an excellent idea of which of its children are the most important (or time critical) and which are the least. Unfortunately, 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.

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. Let us consider the database example once again. 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 in detail 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.

2.28 Thread Scheduling

User-level thread scheduling

Since there are no clock interrupts to multiprogram threads, a thread may continue running as long as it wants to. If it uses up the process' entire quantum, the kernel will select another process to run.

Alternatively, a thread may yield the CPU back to other threads after running for a while.

Kernel-level thread scheduling

Here the kernel picks a particular thread to run. It does not have to take into account which process the thread belongs to, but it can if it wants to. The thread is given a quantum and is forceably suspended if it exceeds the quantum.

Difference between user-level and kernel-level thread scheduling

1. **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.
2. With kernel-level threads, having a thread block on I/O does not suspend the entire process as it does with user-level threads.

2.29 Exponential Average Algorithm (*Algorithm for estimating the running time of a process*)

Let, for some process,

T_0 = The *estimated* run time / CPU burst length
T_1 = The *actual* run time / CPU burst length measured

We can update our estimate for the next run time / CPU burst length (t_2) by taking the *average* of these two numbers, that is,

$$T_2 = \frac{T_1 + T_0}{2}$$

More generally, we can take the *weighted sum* of those two numbers, that is,

$$T_2 = \alpha T_1 + (1 - \alpha) T_0$$

where α is a constant ($0 \le \alpha \le 1$).

Now, to estimate the $(n + 1)^{th}$ run time, let,

t_n = The actual n^{th} run time / CPU burst length measured
τ_n = The run time / CPU burst which was estimated for this n^{th} run time / CPU burst
τ_{n+1} = The estimated value for the *next* run time / CPU burst

∴ For $0 \le \alpha \le 1$, we have

$$\boldsymbol{\tau_{n+1} = \alpha\, t_n + (1 - \alpha)\, \tau_n}$$

The value of t_n contains our most recent information.

τ_n stores the past history.

The parameter α controls the relative weight of recent and past history in our prediction.

If $\alpha = 0$, then $\tau_{n+1} = \tau_n$ and recent history has no effect.

If $\alpha = 1$, then $\tau_{n+1} = t_n$ and only the most recent CPU burst matters (history is assumed to be old and irrelevant).

More commonly, $\alpha = ½$, so recent history and past history are *equally weighted*.

The initial estimated run time τ_0 can be defined as a constant or as an overall system average.

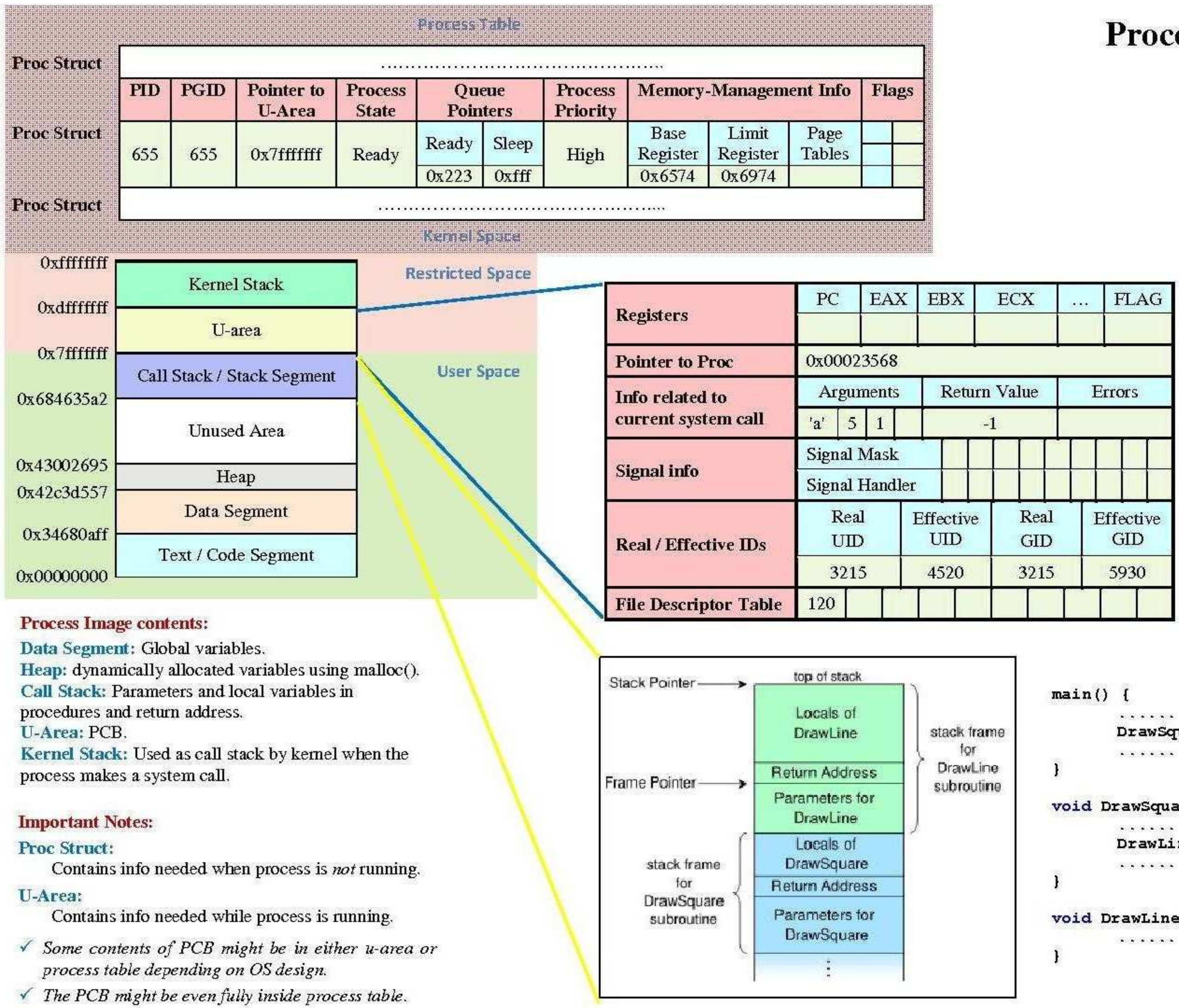
Now, to expand this formula up to the initial estimated run time τ_0, we substitute τ_n with its value:

$$\begin{aligned}
\tau_{n+1} &= \alpha\, t_n + (1 - \alpha)\, \tau_n \\
\therefore\ \tau_n &= \alpha\, t_{n-1} + (1 - \alpha)\, \tau_{n-1} \\
\therefore\ \tau_{n-1} &= \alpha\, t_{n-2} + (1 - \alpha)\, \tau_{n-2}
\end{aligned}$$

$$\begin{aligned}
\tau_{n+1} &= \alpha\, t_n + (1 - \alpha)\, \tau_n \\
&= \alpha\, t_n + (1 - \alpha)\, \{\alpha\, t_{n-1} + (1 - \alpha)\, \tau_{n-1}\} \\
&= \alpha\, t_n + (1 - \alpha)\, \alpha\, t_{n-1} + (1 - \alpha)^2\, \tau_{n-1} \\
&= \alpha\, t_n + (1 - \alpha)\, \alpha\, t_{n-1} + (1 - \alpha)^2\, \{\alpha\, t_{n-2} + (1 - \alpha)\, \tau_{n-2}\} \\
&= \alpha\, t_n + (1 - \alpha)^1\, \alpha\, t_{n-1} + (1 - \alpha)^2\, \alpha\, t_{n-2} + (1 - \alpha)^3\, \tau_{n-2} \\
&= \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\
&= \alpha\, t_n + (1 - \alpha)\, \alpha\, t_{n-1} + \ldots + (1 - \alpha)^{j}\, \alpha\, t_{n-j} + \ldots + (1 - \alpha)^{n+1}\, \tau_0
\end{aligned}$$

The technique of estimating the next value in a series by taking the weighted average of the *current measured value* and the *previous estimate* is sometimes called ***aging***. It is applicable to many situations where a prediction must be made based on previous values.

Process Structure



Process Table
Proc Struct
Proc Struct
Proc Struct
PID | PGID | Pointer to U-Area | Process State | Queue Pointers | Process Priority | Memory-Management Info | Flags
655 | 655 | 0x7fffffff | Ready | Ready 0x223 | Sleep 0xfff | High | Base Register 0x6574 | Limit Register 0x6974 | Page Tables
Kernel Space
Restricted Space
User Space
0xffffffff
Kernel Stack
0xdfffffff
U-area
0x7fffffff
Call Stack / Stack Segment
0x684635a2
Unused Area
0x43002695
Heap
0x42c3d557
Data Segment
0x34680aff
Text / Code Segment
0x00000000
Registers: PC, EAX, EBX, ECX, …, FLAG
Pointer to Proc: 0x00023568
Info related to current system call: Arguments 'a' 5 1; Return Value -1; Errors
Signal info: Signal Mask; Signal Handler
Real / Effective IDs: Real UID 3215; Effective UID 4520; Real GID 3215; Effective GID 5930
File Descriptor Table: 120
Stack Pointer
top of stack
Locals of DrawLine
Return Address
Parameters for DrawLine
stack frame for DrawLine subroutine
Frame Pointer
Locals of DrawSquare
Return Address
Parameters for DrawSquare
stack frame for DrawSquare subroutine

Process Image contents:

Data Segment: Global variables.
Heap: dynamically allocated variables using malloc().
Call Stack: Parameters and local variables in procedures and return address.
U-Area: PCB.
Kernel Stack: Used as call stack by kernel when the process makes a system call.

Important Notes:

Proc Struct:
Contains info needed when process is *not* running.

U-Area:
Contains info needed while process is running.

✓ *Some contents of PCB might be in either u-area or process table depending on OS design.*

✓ *The PCB might be even fully inside process table.*

```
main() {
        ......
        DrawSquare();
        ......
}

void DrawSquare(x1, y1, x2, y2) {
        ......
        DrawLine();
        ......
}

void DrawLine(x1, y1, x2, y2) {
        ......
}
```

CHAPTER 3
DEADLOCK

Roadmap and Concepts in brief

- In this chapter, we'll learn how operating system can handle deadlocks.
- What is deadlock?

  A set of processes is said to be deadlocked if each process in the set is waiting for an event that only another process in the set can cause.
- When does deadlock occur? Or, what are the conditions for deadlock to occur?
  1. **Mutual exclusion condition:** Each resource is either currently assigned to *exactly one* process or is available.
  2. **Hold and wait condition:** Processes currently holding resources granted earlier can request new resources.
  3. **No preemption condition:** Resources previously granted cannot be forcibly taken away from a process. They must be explicitly released by the process holding them.
  4. **Circular wait condition:** There must be a circular chain of two or more processes, each of which is waiting for a resource held by the next member of the chain.
- How can deadlocks be handled?
  1. **Just ignore the problem altogether. (The Ostrich Algorithm)**
  2. **Detection and recovery.** Let deadlocks occur, detect them, and take action.

     Deadlock detection
     - **Detection with single instance of each resource type/class**

       A resource-allocation graph is checked to see whether cycles exist in there.
     - **Detection with multiple instances of each resource type/class**

       A matrix-based algorithm is used to determine whether any unmarked processes exist.

     Deadlock Recovery
     - **Recovery through terminating / killing processes**
       1. Abort all deadlocked processes.
       2. Abort one process at a time until the deadlock cycle is eliminated.
       3. A process *not* in the cycle can be chosen as the victim in order to release its resources.
     - **Recovery through resource preemption**
     - **Recovery through rollback**
  3. **Prevention**, by structurally negating one of the four conditions necessary to cause a deadlock.
     - **Attacking the *mutual exclusion* condition:** share all resources.
     - **Attacking the *hold and wait* condition**
       1. Each process is required to request and be allocated all its resources before it begins execution.
       2. Each process is to request resources only when it has none.
     - **Attacking the *no preemption* condition:** Take resources away.
     - **Attacking the *circular wait* condition**
       1. Each process can request resources only in an *increasing* order of enumeration.
       2. No process can request a resource *lower* than what it is already holding.
  4. **Dynamic avoidance** by careful resource allocation.
     - **The Banker's algorithm for a single resource**
     - **The Banker's algorithm for multiple resources**

Theory

3.1 Deadlock

Deadlock can be defined formally as follows:

A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause.

In most cases, the event that each process is waiting for is the release of some resource currently possessed by another member of the set.

3.2 Resources

Deadlocks can occur when processes have been granted exclusive access to devices, files, and so forth. To make the discussion of deadlocks as general as possible, we will refer to the objects granted as *resources*. A resource can be a hardware device (e.g., a tape drive) or a piece of information (e.g., a locked record in a database). In short, a resource is anything that can be used by only a single process at any instant of time.

Resources come in two types:

1. Preemptable Resources
2. Nonpreemptable Resources

Preemptable Resources

A *preemptable resource* is one that can be taken away from the process owning it with no ill effects.

Memory is an example of a preemptable resource. Suppose, process *A* and process *B* are deadlocked because *A* owns a shared memory and *B* needs it. In this case, it is possible to preempt (take away) the memory from *A* by swapping it out and swapping *B* in. Thus, no deadlock occurs.

Nonpreemptable Resources

A *nonpreemptable resource* is one that cannot be taken away from its current owner without causing the computation to fail.

If a process has begun to burn a CD-ROM, suddenly taking the CD recorder away from it and giving it to another process will result in a garbled CD. CD recorders are not preemptable at an arbitrary moment.

In general, deadlocks involve nonpreemptable resources. Potential deadlocks that involve preemptable resources can usually be resolved by reallocating resources from one process to another. Thus our treatment will focus on nonpreemptable resources.

3.3 Conditions for Deadlock

Coffman et al. (1971) showed that *four* conditions *must* hold for there to be a deadlock:

1. **Mutual exclusion condition:** Each resource is either currently assigned to *exactly one* process or is available.

   At least one resource must be held in a non-sharable mode; that is, only *one* process at a time can use the resource. If another process requests that resource, the requesting process must be delayed until the resource has been released.

2. **Hold and wait condition:** Processes currently holding resources granted earlier can request new resources.

   A process must be holding at least one resource and waiting to acquire additional resources that are currently being held by other processes.

3. **No preemption condition:** Resources previously granted cannot be forcibly taken away from a process. They must be explicitly released by the process holding them.

   Resources cannot be preempted; that is, a resource can be released only voluntarily by the process holding it, after that process has completed its task.

4. **Circular wait condition:** There must be a circular chain of two or more processes, each of which is waiting for a resource held by the next member of the chain.

   A set {P_0, P_1, ..., P_n} of waiting processes must exist such that P_0 is waiting for a resource held by P_1, P_1 is waiting for a resource held by P_2, …, P_{n-1} is waiting for a resource held by P_n, and P_n is waiting for a resource held by P_0.

All four of these conditions *must* be present for a deadlock to occur. If one of them is absent, no deadlock is possible.

3.4 Deadlock Modeling / Deadlock Characterization using Resource-Allocation Graphs

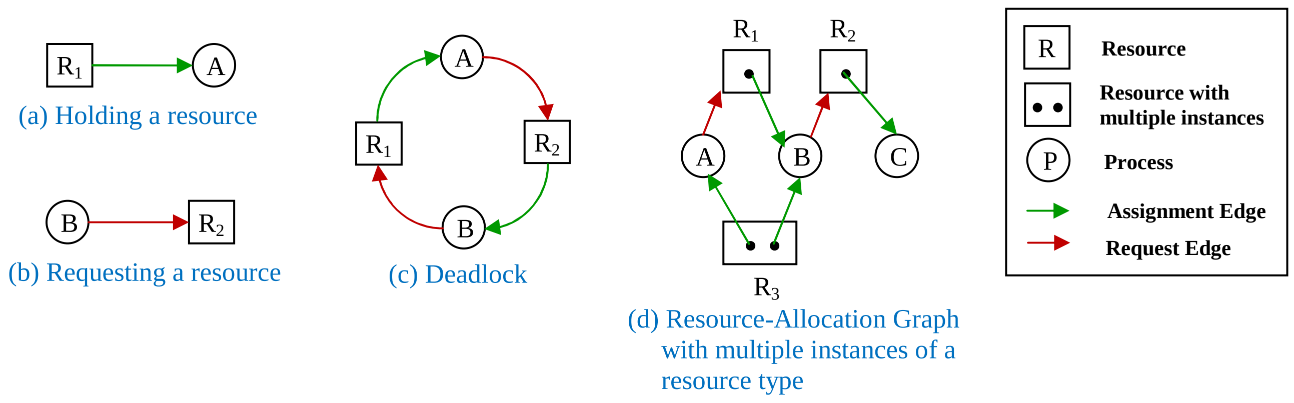

R1
A
(a) Holding a resource
B
R2
(b) Requesting a resource
A
R1
R2
B
(c) Deadlock
R1
R2
A
B
C
R3
(d) Resource-Allocation Graph with multiple instances of a resource type
R
Resource
Resource with multiple instances
P
Process
Assignment Edge
Request Edge

Figure 3.4a: Resource-Allocation Graph.

Holt (1972) showed how the four conditions for deadlock can be modeled using directed graphs.

In this graph, there are two types of **nodes**:

1. **Processes** – shown as circles
2. **Resources** – shown as squares or rectangles

However, there are also two types of **edges**:

1. **Assignment edge** – a directed edge from a resource to a process
2. **Request edge** - a directed edge from a process to a resource

Resource-allocation graphs can be used for single-instance resources as well as multiple-instance resources.

In resource-allocation graphs for single-instance resources (as shown in *figure 3.4a (a), (b) and (c)*), if a cycle is found, then there exists a deadlock.

In resource-allocation graphs for multiple-instance resources, the following points should be noted:

- The number of instances of a resource is depicted using *dots* inside the rectangle.
- A *request* edge points to only the rectangle; whereas an *assignment* edge must also designate one of the dots in the rectangle (*see figure 3.4a (d)*).
- In case of multiple-instance resources, if a resource-allocation graph does *not* have a cycle, then the system is *not* in a deadlocked state. If there is a cycle, then the system *may* or *may not* be in a deadlocked state. For example, in *figure 3.4b*, there exists a cycle, but no deadlock. In that figure, process *D* may release its instance of resource type R_2. That resource can then be allocated to *B*, breaking the cycle.

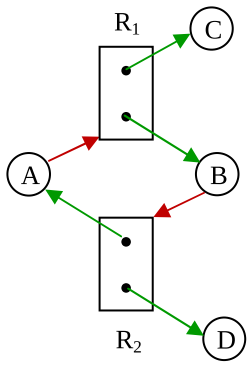


Figure 3.4b: Resource-Allocation Graph with a cycle but no deadlock.

3.5 Methods for Handling Deadlocks

In general, four strategies are used for dealing with deadlocks:

1. **Just ignore the problem altogether.** Maybe if you ignore it, it will ignore you!
2. **Detection and recovery.** Let deadlocks occur, detect them, and take action.
3. **Prevention**, by structurally negating one of the four conditions necessary to cause a deadlock.
4. **Dynamic avoidance** by careful resource allocation.

3.6 The Ostrich Algorithm

The simplest approach is the ostrich algorithm: stick your head in the sand and pretend there is no problem at all!

Advantages

In many systems, deadlocks occur infrequently (say, once per year); thus, this method is cheaper than the prevention, avoidance, or detection and recovery methods, which must be used constantly and hence costly. Most operating systems, including UNIX and Windows, use this approach.

Disadvantages

In this case, we may arrive at a situation where the system is in a deadlocked state yet has no way of recognizing what has happened. The undetected deadlock will result in deterioration of the system's performance, because resources are being held by processes that cannot run; and because more and more processes, as they make requests for resources, will enter a deadlocked state. Eventually, the system will stop functioning and will need to be restarted manually.

3.7 Deadlock Detection and Recovery

When this technique is used, the system does not attempt to prevent deadlocks from occurring. Instead, it lets them occur, tries to detect when this happens, and then takes some action to recover after the fact.

In this environment, the system must provide:

- An algorithm that examines the state of the system to determine whether a deadlock has occurred.
- An algorithm to recover from the deadlock.

Deadlock Detection

Deadlock detection with single instance of each resource type

For such a system, we can construct a resource-allocation graph. If this graph contains one or more cycles, a deadlock exists. Any process that is part of a cycle is deadlocked. If no cycles exist, the system is not deadlocked.

We can run a DFS (or BFS) algorithm to detect cycle in the graph as well as the processes that are part of the deadlock (i.e., the cycle).

Deadlock detection with multiple instances of each resource type

When multiple copies of some of the resources exist, a different approach is needed to detect deadlocks. We will now present a matrix-based algorithm for detecting deadlock among *n* processes, P_1 through P_n.

Let,

$\boldsymbol{m}$	=	number of resource types/classes
$\boldsymbol{E}$	=	**Existing resource vector**, which gives the total number of instances of *each* resource in existence
$\boldsymbol{E_i}$	=	Total number of instances of resource of *i* class
$\boldsymbol{A}$	=	**Available resource vector**, which gives the number of instances currently available for each resource

A_i = The number of instances of resource *i* that are currently available

C = **Current allocation matrix**. The *i*-th row of *C* tells how many instances of each resource class P_i currently holds. Thus C_{ij} is the number of instances of resource *j* that are held by process *i*.

R = **Request matrix**. The *i*-th row of *R* tells how many instances of each resource class P_i needs. Thus R_{ij} is the number of instances of resource *j* that P_i wants.

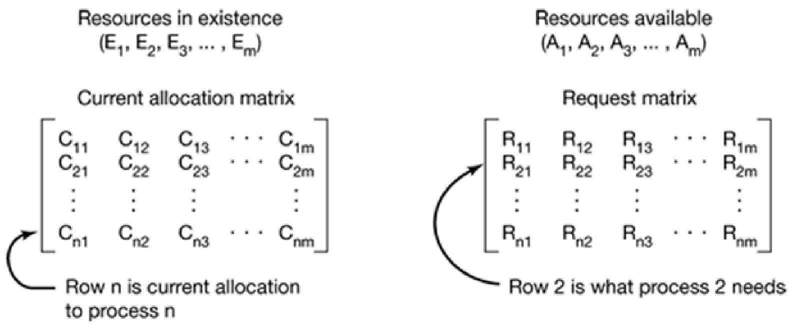

Resources in existence
$(E_1, E_2, E_3, \ldots, E_m)$
Resources available
$(A_1, A_2, A_3, \ldots, A_m)$
Current allocation matrix
Request matrix
$$\begin{bmatrix} C_{11} & C_{12} & C_{13} & \cdots & C_{1m} \\ C_{21} & C_{22} & C_{23} & \cdots & C_{2m} \\ \vdots & \vdots & \vdots & & \vdots \\ C_{n1} & C_{n2} & C_{n3} & \cdots & C_{nm} \end{bmatrix}$$
$$\begin{bmatrix} R_{11} & R_{12} & R_{13} & \cdots & R_{1m} \\ R_{21} & R_{22} & R_{23} & \cdots & R_{2m} \\ \vdots & \vdots & \vdots & & \vdots \\ R_{n1} & R_{n2} & R_{n3} & \cdots & R_{nm} \end{bmatrix}$$
Row n is current allocation to process n
Row 2 is what process 2 needs

Figure 3.7: The four data structures needed by the deadlock detection algorithm.

An important invariant holds for these four data structures. In particular, every resource is either allocated or is available. This observation means that

$$\sum_{i=1}^{n} C_{ij} + A_j = E_j$$

In other words, if we add up all the instances of the resource *j* that have been allocated and to this add all the instances that are available, the result is the number of instances of that resource class that exist.

The deadlock detection algorithm is based on comparing vectors. Let us define the relation $A \leq B$ on two vectors *A* and *B* to mean that each element of *A* is less than or equal to the corresponding element of *B*. Mathematically, $\boldsymbol{A \leq B}$ **holds if and only if** $\boldsymbol{A_i \leq B_i}$ **for** $\boldsymbol{1 \leq i \leq m}$.

Each process is initially said to be unmarked. As the algorithm progresses, processes will be marked, indicating that they are able to complete and are thus not deadlocked. When the algorithm terminates, any unmarked processes are known to be deadlocked.

The deadlock detection algorithm can now be given, as follows:

1. Look for an unmarked process, P_i, for which the *i*-th row of *R* is less than or equal to *A*.
2. If such a process is found, *add* the *i*-th row of *C* to *A*, mark the process, and go back to step 1.
3. If no such process exists, the algorithm terminates.

When the algorithm finishes, all the unmarked processes, if any, are deadlocked.

When to look for deadlocks

Now that we know how to detect deadlocks, the question of when to look for them comes up. Some possibilities include:

1. Check every time a resource request is made.

   Advantage: Detects deadlocks as early as possible.

   Disadvantage: Potentially expensive in terms of CPU time.

2. Check every *k* minutes.
3. Check only when the CPU utilization has dropped below some threshold[17].

   The reason for considering the CPU utilization is that if enough processes are deadlocked, there will be few runnable processes, and the CPU will often be idle.

[17] **Threshold:** A region marking a boundary.

Deadlock Recovery

Recovery through terminating / killing processes

1. **Abort all deadlocked processes.**

   Disadvantage: The *expense* is high. The deadlocked processes may have computed for a long time, and the results of these partial computations must be discarded and probably will have to be recomputed later.

2. **Abort one process at a time until the deadlock cycle is eliminated.**

   Disadvantage: The *overhead* is high. Because, after each process is aborted, a deadlock-detection algorithm must be invoked to determine whether any processes are still deadlocked.

3. **A process *not* in the cycle can be chosen as the victim in order to release its resources.**

   For example, one process might hold a printer and want a plotter, with another process holding a plotter and wanting a printer. These two are deadlocked. A third process may hold another identical printer and another identical plotter and be happily running. Killing the third process will release these resources and break the deadlock involving the first two.

Recovery through resource preemption

In some cases it may be possible to temporarily take a resource away from its current owner and give it to another process. The example for this (the resource being *memory*) has been discussed in *theory 3.2* during the discussion of *preemptable resources*.

However, the ability to take a resource away from a process, have another process use it, and then give it back without the process noticing it is highly dependent on the nature of the resource. Recovering this way is frequently difficult or impossible. Choosing the process to suspend depends largely on which ones have resources that can easily be taken back.

Recovery through rollback

If we preempt a resource from a process, what should be done with that process? Clearly, it cannot continue with its normal execution; it is missing some needed resource. We must roll back the process to some safe state and restart it from that state.

Since, in general, it is difficult to determine what a safe state is, the simplest solution is a total rollback: Abort the process and then restart it. Although it is more effective to roll back the process only as far as necessary to break the deadlock, this method requires the system to keep more information about the state of all running processes.

3.8 Deadlock Prevention

Deadlock prevention provides a set of methods for ensuring that at least one of the necessary conditions for occurring deadlock cannot hold. These methods prevent deadlocks by constraining how requests for resources can be made. We elaborate on this approach by examining each of the four necessary conditions separately.

Attacking the *Mutual Exclusion* Condition

If no resource were ever assigned *exclusively* to a single process, we would never have deadlocks. Sharable resources do not require mutually exclusive access and thus cannot be involved in a deadlock. Read-only files are a good example of a sharable resource. If several processes attempt to open a read-only file at the same time, they can be granted simultaneous access to the file.

In general, however, we cannot prevent deadlocks by denying the mutual-exclusion condition, because some resources (for example, CD-ROM drives) are intrinsically[18] non-sharable.

Attacking the *Hold and Wait* Condition

To ensure that the hold-and-wait condition never occurs in the system, we must guarantee that, whenever a process requests a resource, it does not hold any other resources. There are two ways to

[18] **Intrinsically:** With respect to its inherent nature.

make this possible:

1. **Each process is required to request and be allocated all its resources before it begins execution.**

   Disadvantages:

   1. Many processes do not know how many resources they will need until they have started running.
   2. Resources will not be used optimally with this approach.

      Take, as an example, a process that reads data from an input tape, analyzes it for an hour, and then writes an output tape as well as plotting the results. If all resources must be requested in advance, the process will tie up the output tape drive and the plotter for an hour.

2. **Each process is to request resources *only* when it has *none*.**

   A process may request some resources and use them. Before it can request any additional resources, however, it must temporarily release all the resources that it is currently allocated. After that, it will try to get everything it needs all at once.

   Disadvantage: starvation is possible. A process that needs several popular resources may have to wait indefinitely, because at least one of the resources that it needs is always allocated to some other process.

Attacking the *No Preemption* Condition

This technique is often applied to resources whose state can be easily saved and restored later, such as CPU registers and memory space. It cannot generally be applied to such resources as printers and tape drives. Just think about this situation: a process has been assigned a printer and is in the middle of printing its output; but right at that moment the printer is forcibly taken away because a needed plotter is not available!

Attacking the *Circular Wait* Condition

The fourth and final condition for deadlocks is the circular-wait condition. One way to ensure that this condition never holds is to impose a total ordering of all resource types and to require that each process requests resources in an increasing order of enumeration.

For example, we can provide a global numbering of all the resources, as shown in *figure 3.8(a)*. Now, there might be two protocols to be followed:

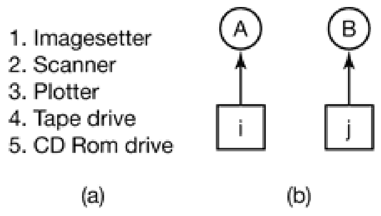


Figure 3.8: (a) Numerically ordered resources. **(b)** A resource graph.

1. **Each process can request resources only in an *increasing* order of enumeration.**

   A process may request first a scanner (no. 2) and then a tape drive (no. 4), but it may not request first a plotter (no. 3) and then a scanner (no. 2).

   Proof that with this rule, the resource allocation graph can never have cycles:

   Let us see why this is true for the case of two processes, in *figure 3.8(b)*. We can get a deadlock only if *A* requests resource *j* and *B* requests resource *i*. Assuming *i* and *j* are distinct resources, they will have different numbers. If $i > j$, then *A* is not allowed to request *j* because that is lower than what it already has. If $i < j$, then *B* is not allowed to request *i* because that is lower than what it already has. Either way, deadlock is impossible.

2. **No process can request a resource *lower* than what it is already holding.**

   For example, if a process is holding resources 2 and 4, it cannot request for resource 3 until it releases resource 4. However, it might release all of its resources and then request for any resource in the list.

Disadvantage of attacking the circular wait condition

1. It may be impossible to find an ordering that satisfies everyone.
2. When the resources include process table slots, disk spooler space, locked database records, and other abstract resources, the number of potential resources and different uses may be so large that no ordering could possibly work.

Summary of approaches to deadlock prevention

Condition	Approach
Mutual exclusion	Share everything
Hold and wait	Request all resources initially
No preemption	Take resources away
Circular wait	Order resources numerically

3.9 Deadlock Avoidance

Deadlock avoidance requires that the operating system be given in advance additional information concerning *which resources* a process will request and use during its lifetime. With this additional knowledge, it can decide for each request whether or not the process should wait. To decide whether the current request can be satisfied or must be delayed, the system must consider the resources currently available, the resources currently allocated to each process, and the future requests and releases of each process.

Safe and Unsafe States

At any instant of time, there is a current state consisting of *E*, *A*, *C*, and *R*. A state is said to be safe if it is not deadlocked and there is some scheduling order in which every process can run to completion even if all of them suddenly request their maximum number of resources immediately.

For example, let, at time t_0, the number of currently assigned resources of three processes and their total needed resources are as follows:

Process	Currently Assigned	Maximum Needs
P_0	5	10
P_1	2	4
P_2	2	9

Suppose that the system has 12 resources in total. Now, at time t_0, the system is in a safe state. Because, the sequence $<P_1, P_0, P_2>$ satisfies the safety condition.

At time t0, the system has 3 free resources. So, P_1 can be allocated all its resources. After P1 finishes its task, it would return all the resources it had and the number of free resources then becomes 5. Now, P_0 can be served, and similarly, after it, P_2 can be served. Thus, all the processes can complete without deadlock.

If, however, P1 needed a maximum of 6 resources, then the state would have become unsafe. Deadlock would occur if no other processes released their resources. But if any process releases its resources, no deadlock will occur even if the system is in unsafe state.

A safe state is not a deadlocked state. Conversely, a deadlocked state is an unsafe state. Not all unsafe states are deadlocks, however. An unsafe state *may* lead to a deadlock. It depends on whether the other processes release the resources held by them.

Safe Sequence

A sequence of processes $<P_1, P_2, ..., P_n>$ is a safe sequence for the current allocation state if, for each P_i, the resource requests that P_i can still make can be satisfied by the currently available resources plus the resources held by all P_j, with $j < i$. In this situation, if the resources that P_i needs are not immediately available, then P_i can wait until all P_j have finished. When they have finished, P_i can obtain all of its needed resources, complete its designated task, return its allocated resources, and terminate. When P_i terminates, P_{i+1} can obtain its needed resources, and so on. If no such sequence exists, then the system state is said to be unsafe.

The Banker's Algorithm

A scheduling algorithm that can avoid deadlocks is due to Dijkstra (1965) and is known as the *banker's algorithm* and is an extension of the deadlock detection algorithm given in *theory 3.7*.

The name was chosen because the algorithm could be used in a banking system to ensure that the bank *never* allocated its available cash in such a way that it could no longer satisfy the needs of *all* its customers.

The Banker's Algorithm for a single resource type

The banker's algorithm considers each request as it occurs, and sees if granting it leads to a safe state. If it does, the request is granted; otherwise, it is postponed until later. To see if a state is safe, the banker checks to see if he has enough resources to satisfy some customer. If so, those loans are assumed to be repaid, and the customer now closest to the limit is checked, and so on. If all loans can eventually be repaid, the state is safe and the initial request can be granted.

The Banker's Algorithm for multiple resource types

This algorithm is just like the deadlock detection algorithm. The difference is that, deadlock detection algorithm is used to detect whether a deadlock *has already* occurred; whereas the Banker's algorithm is used to determine whether a deadlock *will* occur. In the previous case, after the algorithm finishes, if there remain unmarked processes, we can conclude that deadlock has occurred and those processes are part of the deadlock. In the latter case, after the algorithm finishes, if there remain unmarked processes, we can conclude that the initial state was *not* safe and deadlock *might* occur.

Drawback of the Banker's Algorithm

Although in theory the algorithm is wonderful, in practice it is essentially useless. Because:

1. Processes rarely know in advance what their maximum resource needs will be.
2. The number of processes is not fixed, but dynamically varying as new users log in and out.
3. Resources that were thought to be available can suddenly vanish (tape drives can break).

Thus in practice, few, if any, existing systems use the banker's algorithm for avoiding deadlocks.

3.10 Two-phase locking

This is a special-purpose algorithm for avoiding deadlocks, and is used in database systems. In many database systems, an operation that occurs frequently is requesting locks on several records and then updating all the locked records. When multiple processes are running at the same time, there is a real danger of deadlock.

The approach often used is called *two-phase locking*. In the first phase, the process tries to lock all the records it needs, one at a time. If it succeeds, it begins the second phase, performing its updates and releasing the locks. No real work is done in the first phase.

If during the first phase, some record is needed that is already locked, the process just releases all its locks and starts the first phase all over. In a certain sense, this approach is similar to requesting all the resources needed in advance, or at least before anything irreversible is done.

However, this strategy is not applicable in general, because:

1. In real-time systems and process control systems, for example, it is not acceptable to just terminate a process partway through because a resource is not available and start all over again.
2. Neither is it acceptable to start over if the process hits read or written messages to the network, updated files, or anything else that cannot be safely repeated.
3. The algorithm works only in those situations where the programmer has very carefully arranged things so that the program can be stopped at any point during the first phase and restarted. Many applications cannot be structured this way.

3.12 Non-resource Deadlocks

Deadlocks can also occur in situations that do not involve resources at all. Semaphores are a good example. If *down*s on semaphores are done in the wrong order, deadlock can result.

3.13 Starvation

A problem closely related to deadlock is *starvation*. In a dynamic system, requests for resources happen all the time. Some policy is needed to make a decision about who gets which resource when. This policy, although seemingly reasonable, may lead to some processes never getting service even though they are not deadlocked.

As an example, consider allocation of the printer. Imagine that the system uses some kind of algorithm to ensure that allocating the printer does not lead to deadlock. Now suppose that several processes all want it at once. Which one should get it?

One possible allocation algorithm is to give it to the process with the smallest file to print (assuming this information is available). This approach maximizes the number of happy customers and seems fair. Now consider what happens in a busy system when one process has a huge file to print. Every time the printer is free, the system will look around and choose the process with the shortest file. If there is a constant stream of processes with short files, the process with the huge file will never be allocated the printer. It will simply starve to death (be postponed indefinitely, even though it is not blocked).

Starvation can be avoided by using a first-come, first-serve, resource allocation policy.

CHAPTER 4
MEMORY MANAGEMENT

Roadmap and Concepts in Brief

- In this chapter, we'll learn how operating system manages memory.
- First of all, why do we need memory management?

  Ideally, what every programmer would like is an infinitely large, infinitely fast memory that is also nonvolatile, that is, does not lose its contents when the electric power fails. While we are at it, why not also ask for it to be inexpensive, too? Unfortunately technology does not provide such memories. Consequently, most computers have a memory hierarchy, with a small amount of very fast, expensive, volatile cache memory, tens of megabytes of medium-speed, medium-price, volatile main memory (RAM), and tens or hundreds of gigabytes of slow, cheap, nonvolatile disk storage. It is the job of the operating system to coordinate how these memories are used.
- So, what is a memory manager and what is its job?

  The part of the operating system that manages the memory hierarchy is called the *memory manager*. Its job is to keep track of which parts of memory are in use and which parts are not in use, to allocate memory to processes when they need it and deallocate it when they are done, and to manage swapping between main memory and disk when main memory is too small to hold all the processes.
- Well, then how can memory be managed?

  Memory can be managed primarily in two ways:

  1. By moving processes back and forth between main memory and disk during execution.
     a. **Swapping:** Consists of bringing in each process in its entirety, running it for a while, then putting it back on the disk.
     b. **Virtual Memory:** Allows programs to run even when they are only partially in main memory.
        i. **Paging:** Divides a program into several equal-sized blocks.
        ii. **Segmentation:** Divides a program according to the view of user.
  2. By not moving processes between main memory and disk during execution.
     a. Monoprogramming without swapping or paging.
     b. Multiprogramming with fixed partitions.
- Virtual memory has two issues – page tables might be very large, and the mapping must be very fast. How can these issues be addressed?

  Multilevel page tables are used to avoid storing large page tables in memory, and a CPU cache called TLB is used for fast mapping.
- In virtual memory, a heavily used frame might be replaced. How can we ensure replacing a frame which is *not* heavily used?

  There are many algorithms for page replacement. They are:

  1. **The Optimal Page Replacement Algorithm (OPT / MIN)**
     The page that will not be used for the longest period of time is to be replaced.
  2. **Not Recently Used (NRU)**

     When a page fault occurs, the operating system inspects all the pages and divides them into four categories based on the current values of their *R* and *M* bits:

     Class 0: not referenced, not modified.
     Class 1: not referenced, modified.
     Class 2: referenced, not modified.
     Class 3: referenced, modified.

     The NRU algorithm removes a page at random from the *lowest* numbered *nonempty* class.

3. **First-In, First-Out (FIFO)**

   The operating system maintains a list of all pages currently in memory, with the page at the head of the list the oldest one and the page at the tail the most recent arrival. On a page fault, the page at the head is removed and the new page added to the tail of the list.

4. **Second Chance**

   Inspect the *R* bit of the oldest page. If it is 0, the page is both old and unused, so it is replaced immediately. If the *R* bit is 1, the bit is cleared, the page is put onto the end of the list of pages, and its load time is updated. Then the search continues.

   5. **Second Chance (Clock) / Clock Page Replacement**

      All the page frames are kept on a circular linked list in the form of a clock, and a hand points to the *oldest* page.

6. **Least Recently Used (LRU)**

   Throws out the page that has been unused for the longest time.

   7. **Not Frequently Used (NFU) / Least Frequently Used (LFU)**

      A software counter is associated with each page, initially zero. At each clock interrupt, the operating system scans all the pages in memory. For each page, the *R* bit, which is 0 or 1, is added to the counter. When a page fault occurs, the page with the lowest counter is chosen for replacement.

   8. **Aging Algorithm**

      Same as NFU, except that the counters are each shifted right 1 bit before the *R* bit is added in. The *R* bit is added to the leftmost, rather than the rightmost bit.

9. **Working Set**

   The set of pages that a process is currently using is called its ***working set***. When a page fault occurs, find a page not in the working set and evict it.

- How can frames be allocated to a process?

  1. **Equal allocation:** For *m* frames and *n* processes, each process gets *m* / *n* frames.
  2. **Proportional Allocation:** Available memory is allocated to each process proportional to its size.

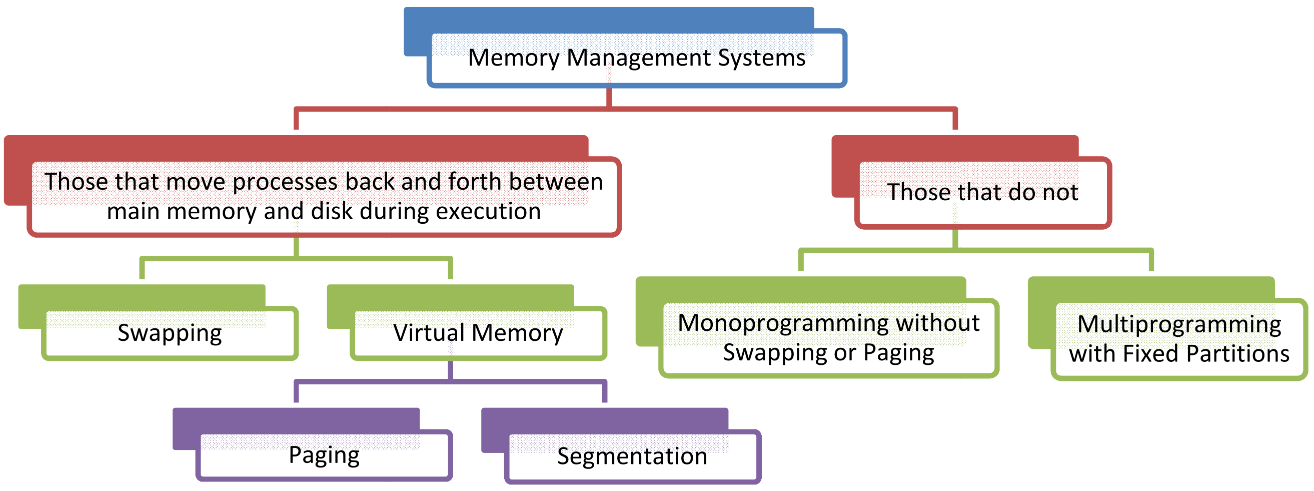
Theory

4.1 The Memory Manager and Its Jobs

The part of the operating system that manages the memory hierarchy is called the *memory manager*. Its jobs are:

1. To keep track of which parts of memory are in use and which parts are not in use.
2. To allocate memory to processes when they need it and deallocate it when they are done.
3. To manage swapping between main memory and disk when main memory is too small to hold all the processes.

4.2 Memory Management Systems


Memory Management Systems
Those that move processes back and forth between main memory and disk during execution
Those that do not
Swapping
Virtual Memory
Monoprogramming without Swapping or Paging
Multiprogramming with Fixed Partitions
Paging
Segmentation

4.3 Monoprogramming Without Swapping or Paging

The simplest possible memory management scheme is to run just one program at a time, sharing the memory between that program and the operating system. Three variations on this theme are shown in the *figure* 4.3.

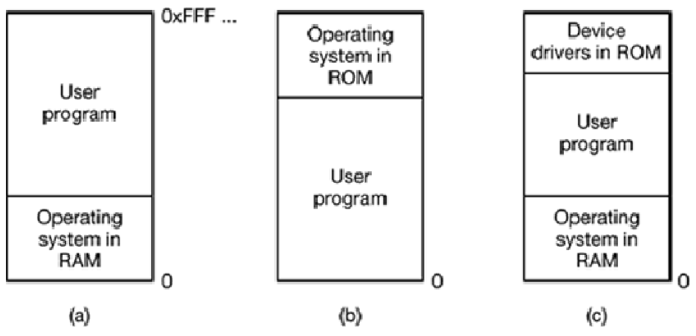


Figure 4.3: Three simple ways of organizing memory with an operating system and one user process. Other possibilities also exist.

4.4 Multiprogramming with Fixed Partitions

The easiest way to achieve multiprogramming is simply to divide memory up into *n* (possibly unequal) partitions. This partitioning can, for example, be done manually when the system is started up.

Implementation Technique 1: Separate Input Queues for Each Partition

When a job arrives, it can be put into the input queue for the smallest partition large enough to hold it.

Disadvantages

1. Since the partitions are fixed in this scheme, any space in a partition not used by a job is lost. This is called *internal fragmentation*.
2. If the queue for a large partition is empty but the queue for a small partition is full, then small jobs have to wait to get into memory, even though plenty of memory is free.

Implementation Technique 2: Single Input Queue for All the Partitions

Algorithm 1:

Whenever a partition becomes free, the job closest to the front of the queue that fits in it could be loaded into the empty partition and run.

Disadvantage: A large partition could be wasted on a small job.

Algorithm 2:

Search the whole input queue whenever a partition becomes free and pick the largest job that fits.

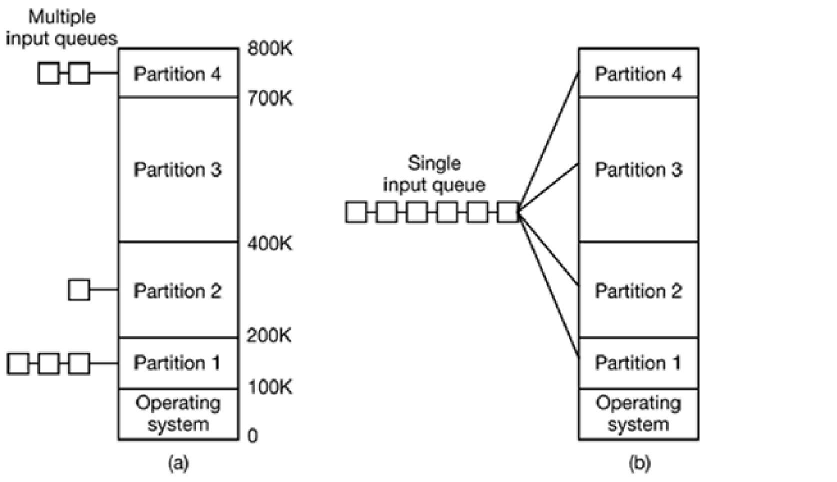


Figure 4.4: (a) Fixed memory partitions with separate input queues for each partition. (b) Fixed memory partitions with a single input queue.

Disadvantage: This algorithm discriminates against small jobs as being unworthy of having a whole partition, whereas usually it is desirable to give the smallest jobs (often interactive jobs) the best service, not the worst.

Algorithm 3:

Have a rule stating that a job that is eligible to run may not be skipped over more than *k* times. Each time it is skipped over, it gets one point. When it has acquired *k* points, it may not be skipped again.

4.5 Issues with Multiprogramming

Modeling Multiprogramming – How many processes should be in memory at a time so that the CPU is fully utilized?

Let,

The *fraction* of time a process waits for its I/O to complete = p

Number of processes in memory *at a time* = n

$\therefore$ The probability that all n processes are waiting for I/O = p^n

$\therefore$ CPU idle time = p^n

$\therefore$ CPU utilization = $1 - p^n$

The *figure* 4.5 shows the CPU utilization as a function of n [n is called the *degree of multiprogramming*].

From the figure, it is clear that if processes spend 80 percent of their time waiting for I/O, at least 10 processes must be in memory at once to get the CPU waste below 10 percent.

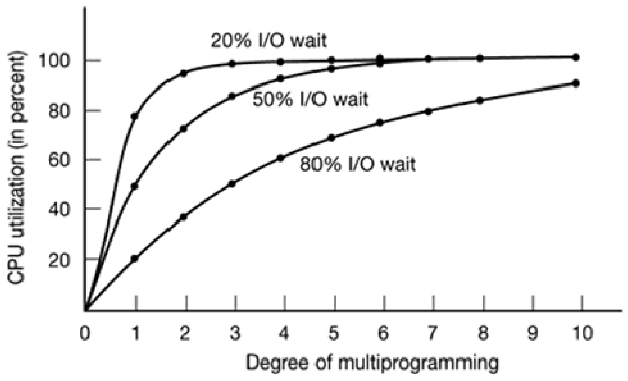


Figure 4.5: CPU utilization as a function of the number of processes in memory.

Relocation and Protection

Multiprogramming introduces two essential problems that must be solved—relocation and protection.

Relocation

From *figure* 4.4, it is clear that different jobs will be run at different addresses. However, during linking, it is assumed that programs will start from the address 00000000. Then, how would the actual addresses of different programs can be found?

Protection

A malicious process might go for writing into other process's memory image. How can it be prevented?

Solution to both relocation and protection problems

Equip the machine with two special hardware registers, called the **base** and **limit** registers. When a process is scheduled, the base register is loaded with the address of the start of its partition, and the limit register is loaded with the length of the partition. Every memory address generated automatically has the base register contents added to it before being sent to memory. Thus if the base register contains the value 100K, a CALL 100 instruction is effectively turned into a CALL 100K+100 instruction, without the instruction itself being modified. Addresses are also checked against the limit register to make sure that they do not attempt to address memory outside the current partition. The hardware protects the base and limit registers to prevent user programs from modifying them.

4.6 Swapping

Swapping consists of bringing in each process *in its entirety*, running it for a while, then putting it back on the disk.

Why *Swapping* Rather Than *Multiprogramming with Fixed Partitions*?

Multiprogramming with fixed partitions is ideal for batch systems. As long as enough jobs can be kept in memory to keep the CPU busy all the time, there is no reason to use anything more complicated.

However, in timesharing or interactive systems, the situation is different. Sometimes there is not enough main memory to hold all the currently active processes, so excess processes must he kept on disk and brought in to run *dynamically*. That's why swapping is needed.

Operation of Swapping

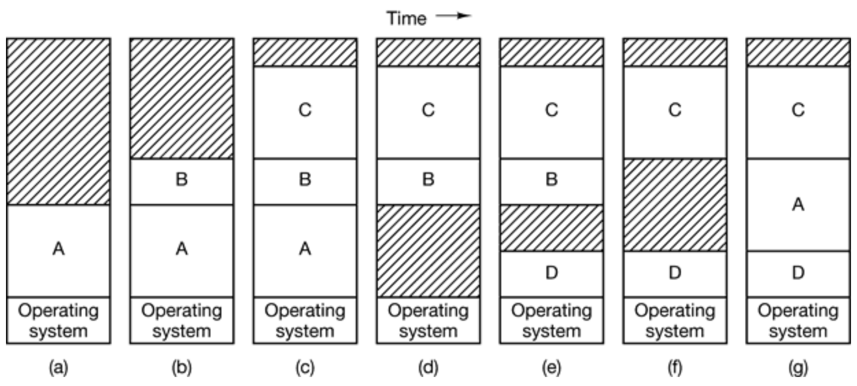


Figure 4.6.1: Memory allocation changes as processes come into memory and leave it. The shaded regions are unused memory.

The operation of a swapping system is illustrated in *figure* 4.6.1. Initially only process *A* is in memory. Then processes *B* and *C* are created or swapped in from disk. In *figure* 4.6.1 (d), *A* is swapped out to disk. Then *D* comes in and *B* goes out. Finally *A* comes in again. Since *A* is now at a different location, addresses contained in it must be relocated, either by software when it is swapped in or (more likely) by hardware during program execution.

Difference Between *Fixed Partitions* and *Variable Partitions*

The main difference between the fixed partitions of *figure* 4.4 and the variable partitions of *figure* 4.6.1 is that the number, location, and size of the partitions vary dynamically in the latter as processes come and go, whereas they are fixed in the former.

The flexibility of not being tied to a fixed number of partitions that may be too large or too small improves memory utilization, but it also complicates allocating and deallocating memory, as well as keeping track of it.

How Much Memory Should be Allocated for a Process When it is Created or Swapped in?

If processes are created with a fixed size that never changes, then the allocation is simple: the operating system allocates exactly what is needed, no more and no less.

If it is expected that most processes will grow as they run, it is probably a good idea to allocate a little extra memory whenever a process is swapped in or moved, to reduce the overhead associated with moving or swapping processes that no longer fit in their allocated memory.

In *figure* 4.6.2, we see a memory configuration in which space for growth has been allocated to two processes.

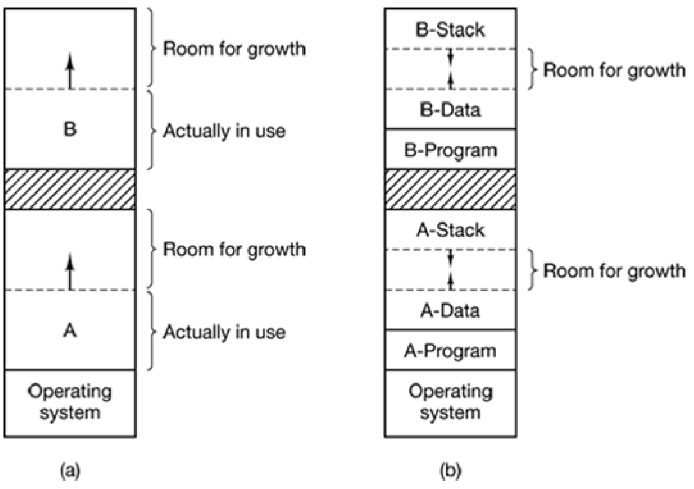


Figure 4.6.2: (a) Allocating space for a single growing segment.
(b) Allocating space for a growing stack and a growing data segment.

Memory Management: Keeping Track of Memory Usage

When memory is assigned dynamically, the operating system must manage it. In general terms, there are two ways to keep track of memory usage: *bitmaps* and *free lists*.

Memory Management with Bitmaps

- Memory is divided up into allocation units, perhaps as small as a few words and perhaps 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).
- The size of the allocation unit is an important design issue. The smaller the allocation unit, the larger the bitmap. If the allocation unit is chosen large, the bitmap will be smaller, but appreciable memory may be wasted in the last unit of the process if the process size is not an exact multiple of the allocation unit.

Advantage:

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.

Disadvantage:

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

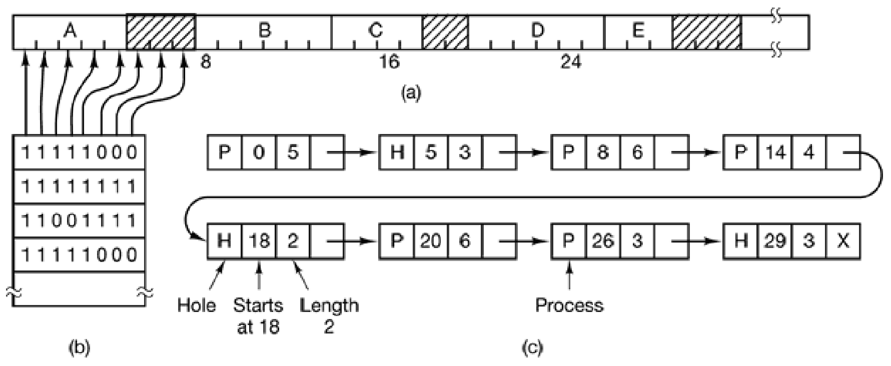


Figure 4.6.3: (a) A part of memory with five processes and three holes. The tick marks show the memory allocation units. The shaded regions (0 in the bitmap) are free.
(b) The corresponding bitmap.
(c) The same information as a list.

Memory Management with Linked Lists

- A linked list of allocated and free memory segments is maintained, where a segment is either a process or a hole between two processes.
- Each entry in the list specifies a hole (H) or process (P), the address at which it starts, the length, and a pointer to the next entry.
- Updating the list is straightforward. A terminating process normally has two neighbors (except when it is at the very top or bottom of memory). These may be either processes or holes, leading to the four combinations of *figure* 4.6.4.

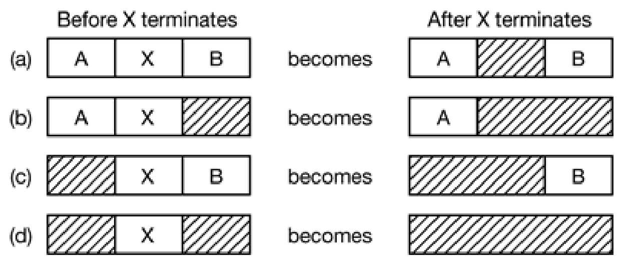


Figure 4.6.4: Four neighbor combinations for the terminating process *X*.

Algorithms for Allocating Memory for Processes

When the processes and holes are kept on a list sorted by address, several algorithms can be used to allocate memory for a newly created process (or an existing process being swapped in from disk). We assume that the memory manager knows how much memory to allocate.

When a Single List is Maintained for Both Processes and Holes

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.

Next Fit

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.

Next fit gives slightly *worse* performance than first fit.

Best Fit

Searches the entire list 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.

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.

Worst Fit

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 hole broken off will be big enough to be useful.

Simulation has shown that worst fit is not a very good idea either.

When Distinct Lists are Maintained for Processes and Holes

All four algorithms can be speeded up by maintaining separate lists for processes and holes. In this way, all of them devote their full energy to inspecting holes, not processes. The inevitable price that is paid for this speedup on allocation is the additional complexity and slowdown when deallocating memory, since a freed segment has to be removed from the process list and inserted into the hole list.

If distinct lists are maintained for processes and holes, the hole list may be kept sorted on size, to make best fit faster. When best fit searches a list of holes from smallest to largest, as soon as it finds a hole that fits, it knows that the hole is the smallest one that will do the job, hence the best fit. No further searching is needed, as it is with the single list scheme. With a hole list sorted by size, first fit and best fit are equally fast, and next fit is pointless.

Quick Fit

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 either be put 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 is possible is expensive. If merging is not done, memory will quickly fragment into a large number of small holes into which no processes fit.

Memory Fragmentation

Internal Fragmentation

Internal fragmentation exists when some portion of a fixed-size partition is not being used by a process.

External Fragmentation

External fragmentation exists when there is enough total memory space to satisfy a request, but the available spaces are not contiguous; storage is fragmented into a large number of small holes.

Solution to External Fragmentation

1. **Compaction:** The goal is to shuffle the memory contents so as to place all free memory together in one large block. It is usually not done because it requires a lot of CPU time.
2. **Paging or Segmentation:** The logical address space of the processes is permitted to be noncontiguous, thus allowing a process to be allocated physical memory wherever the latter is available.

4.7 Virtual Memory

Virtual memory allows programs to run even when they are only partially in main memory.

The basic idea is to split a program into pieces, called ***overlays*** and swap them in and out from memory as needed.

4.8 Paging / Demand Paging

The virtual address space is divided up into units called ***pages***. The corresponding units in the physical memory are called ***page frames***. The pages and page frames are always the same size, and is typically a power of 2.

With demand-paged virtual memory, pages are only loaded when they are demanded during program execution; pages that are never accessed are thus never loaded into physical memory.

Mapping of Pages to Page Frames

Every address generated by the CPU is divided into two parts: a **page number**, ***p*** and a **page offset, *d***. The page number is used as an index into a **page table**. The page table contains the base address of each page in physical memory. This base address is combined with the page offset to define the physical memory address that is sent to the memory unit.

If the size of logical address space is 2^m, and a page size is 2^n addressing units (bytes or words), then the high-order $m - n$ bits of a logical address designate the page number, and the n low-order bits designate the page offset. Thus, the logical address is as follows:

page number	page offset
p	*d*
m - n	*n*

Where *p* is an index into the page table and *d* is the displacement within the page.

Example

In *figure* 4.8.1, virtual address 20500 is 20 bytes from the start of virtual page 5 (virtual addresses 20480 to 24575) and maps onto physical address 12288 + 20 = 12308.

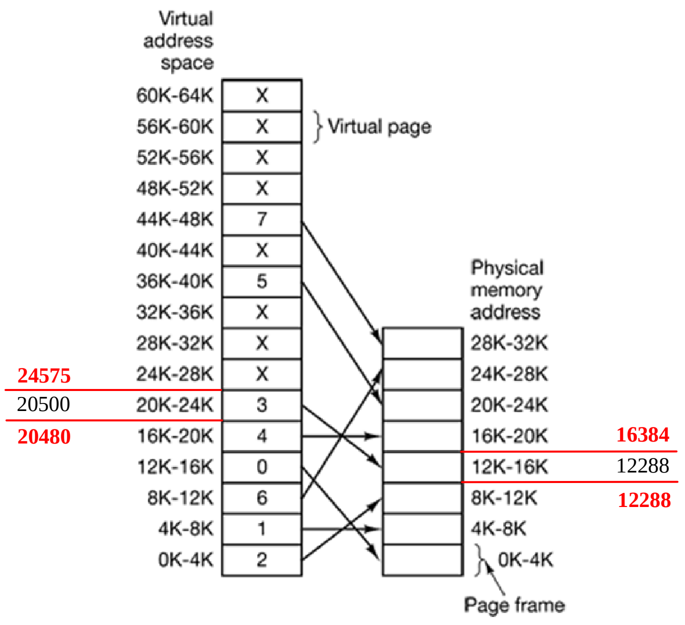


Figure 4.8.1: The relation between virtual addresses and physical memory addresses as is given by the page table.

Keeping Track of Which Pages are Currently Mapped Onto Physical Memory

From *figure* 4.8.1, since we have only eight physical page frames, only eight of the virtual pages are mapped onto physical memory. The others, shown as a cross in the figure, are not mapped. In the actual hardware, a **Present/absent bit** keeps track of which pages are physically present in memory.

How the Paging System / MMU (Memory Management Unit) Works

In *figure* 4.8.2, we see an example of a virtual address, 8196 (0010000000000100 in binary), being mapped using the MMU map of *figure* 4.8.1.

The incoming 16-bit virtual address is split into a 4-bit page number and a 12-bit offset.

With 4 bits for the page number, we can have 16 pages, and with 12 bits for the offset, we can address all 4096 bytes within a page.

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.

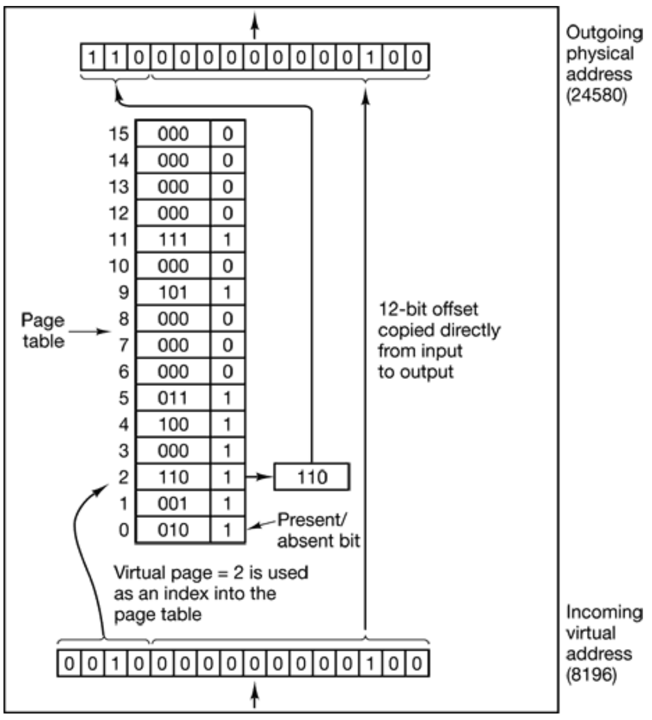


Figure 4.8.2: The internal operation of the MMU with 16 4-KB pages.

Structure of a Page Table Entry

The exact layout of an entry is highly machine dependent, but the kind of information present is roughly the same from machine to machine.

Page frame number

The most important field is the *Page frame number*. After all, the goal of the page mapping is to locale this value.

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.

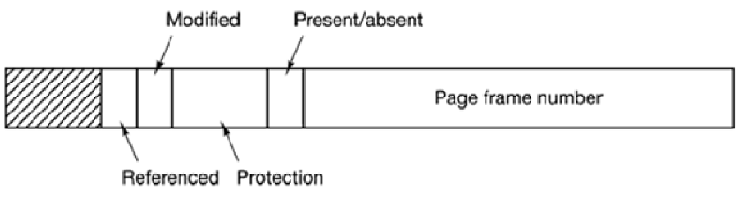


Figure 4.8.3: A typical page table entry.

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.

Modified / Dirty bit

Keeps 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.

Referenced bit

Is set whenever a page is referenced, either for reading or writing. Its value is to help the operating system choose a page to evict when a page fault occurs. Pages that are not being used are 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.

Issues with Paging

Despite the simple description of the paging system, two major issues must be faced:

1. The page table can be extremely large.
2. The mapping must be fast.

Multilevel Page Tables

To get around the problem of having to store huge page tables in memory all the time, many computers use a multilevel page table. A simple example is shown in *figure* 4.8.4.

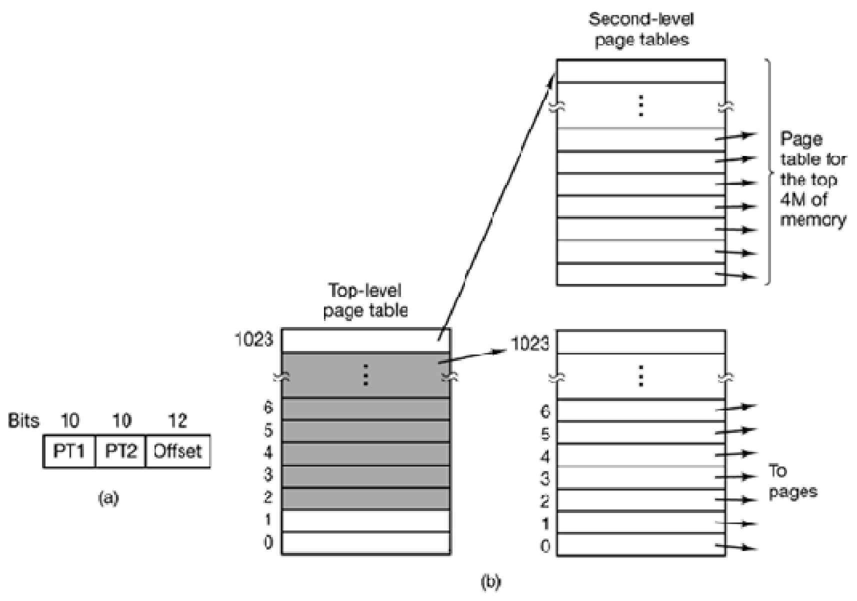


Figure 4.8.4: (a) A 32-bit address with two page table fields. (b) Two-level page tables.

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.

TLB (Translation Lookaside Buffer) / Associative Memory

Most programs tend to make a large number of references to a small number of pages, and not the other way around. Thus, only a small fraction of the page table entries are heavily read; the rest are barely used at all.

Therefore, to speed up the lookup process, computers are equipped 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*, is illustrated in *figure* 4.8.5.

Valid	Virtual page	Modified	Protection	Page frame
1	140	1	RW	31
1	20	0	R X	38
1	130	1	RW	29
1	129	1	RW	62
1	19	0	R X	50
1	21	0	R X	45
1	860	1	RW	14
1	861	1	RW	75

Figure 4.8.5: A TLB to speed up paging.

Structure of a TLB

It is usually inside the MMU and consists of a small number of entries, eight in this example, but rarely more than 64. 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. Another bit indicates whether the entry is valid (i.e., in use) or not.

Working Procedure of TLB

- 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 (i.e., in parallel).
- 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 same way as it would be from the page table itself.
- 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 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. When the TLB is loaded from the page table, all the fields are taken from memory.

TLB Storing ASIDs (Address-Space Identifier)

Some TLBs store *address-space identifiers* (ASIDs) in each TLB entry. An ASID uniquely identifies each process and is used to provide address-space protection for that process. When the TLB attempts to resolve virtual page numbers, it ensures that the ASID for the currently running process matches the ASID associated with the virtual page. If the ASIDs do not match, the attempt is treated as a TLB miss.

In addition to providing address-space protection, an ASID allows the TLB to contain entries for several different processes simultaneously.

If the TLB does not support separate ASIDs, then every time a new page table is selected (for instance, with each context switch), the TLB must be flushed (or erased) to ensure that the next executing process does not use the wrong translation information. Otherwise, the TLB could include old entries that contain valid virtual addresses but have incorrect or invalid physical addresses left over from the previous process.

Effective memory access / reference time

The percentage of times that a particular page number is found in the TLB is called the hit ratio. An 80-percent hit ratio means that we find the desired page number in the TLB 80 percent of the time.

If it takes 20 nanoseconds to search the TLB and 100 nanoseconds to access memory, then a mapped-memory access takes 120 nanoseconds when the page number is in the TLB.

If we fail to find the page number in the TLB (20 nanoseconds), then we must first access memory for the page table and frame number (100 nanoseconds) and then access the desired byte in memory (100 nanoseconds), for a total of 220 nanoseconds.

To find the effective memory-access time, we weight each case by its probability:

$$\textit{effective access time} = 0.80 \times 120 + 0.20 \times 220 = 140 \text{ nanoseconds}$$

Page Fault and How it is Handled

When a process tries to access a page that is not available in main memory, the CPU is caused to trap to the operating system. This trap is called a *page fault*.

Actions taken by the Operating System when a page fault occurs:

1. The page table is checked to determine whether the reference was a valid or an invalid memory access.
2. If the reference was invalid, the process is terminated. If it was valid, but that page is not yet brought in, then it is paged in.
3. A free frame is allocated from the physical memory (by taking one from the free-frame list, or by swapping out another page frame).
4. A disk operation is scheduled to read the desired page into the newly allocated frame.
5. When the disk read is complete, the page table is modified to indicate that the page is now in memory.
6. The instruction that was interrupted by the trap is restarted. The process can now access the page as though it had always been in memory.

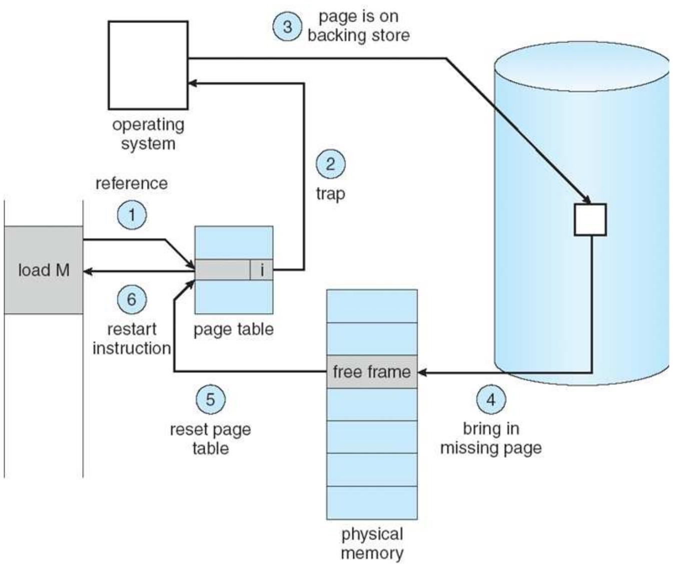


Figure 4.8.6: Actions taken when a page fault occurs.

Performance of Demand Paging

Let p be the probability of a page fault (a.k.a. page fault ratio) ($0 \leq p \leq 1$). We would expect p to be close to zero – that is, we would expect to have only a few page faults. The effective access time is then

$$\text{Effective Memory Access Time} = (1 - p) \times \textit{memory access time} + p \times \textit{page fault service time}$$

To compute the effective access time, we must know how much time is needed to service a page fault. There three major components of the page-fault service time:

1. Service the page-fault interrupt.
2. Read in the page.
3. Restart the process.

The first and third tasks can be reduced, with careful coding, to several hundred instructions. These tasks may take from 1 to 100 microseconds each, and are thus negligible.

The page-switch time, however, will probably be close to 8 milliseconds. A typical hard disk has an average latency of 3 milliseconds, a seek of 5 milliseconds, and a transfer time of 0.05 milliseconds. Thus, the total paging time is about 8 milliseconds, including hardware and software time. Remember also that we are looking at only the device-service time. If a queue of processes is waiting for the device (other processes that have caused page faults), we have to add device-queueing time as we wait for the paging device to be free to service our request, increasing even more the time to swap.

If we take an average page-fault service time of 8 milliseconds and a memory-access time of 200 nanoseconds, then the effective access time in nanoseconds is

$$\begin{aligned}\text{Effective Access Time} &= (1 - p) \times 200 + p \times 8 \text{ ms}\\ &= (1 - p) \times 200 + p \times 8{,}000{,}000\\ &= 200 + 7{,}999{,}800 \times p\end{aligned}$$

We see, then, that the effective access time is directly proportional to the page-fault rate. If one access out of 1,000 causes a page fault, the effective access time is 8.2 microseconds. The computer will be slowed down by a factor of 40 because of demand paging![19] If we want performance degradation to be less than 10 percent, we need

$$200 + 7{,}999{,}800 \times p < 220$$
$$\Rightarrow 7{,}999{,}800 \times p < 20$$
$$\Rightarrow p < \frac{1}{399{,}990}$$

That is, to keep the slowdown due to paging at a reasonable level, we can allow fewer than one memory access out of 399,990 to page-fault.

4.9 Segmentation

A user thinks of a program as a collection of functions and data structures such as arrays, stacks, variables etc. *Segmentation* is a memory-management scheme that supports this user view of memory. A logical address space is a collection of segments. Each segment has a number and a length. The addresses specify both the segment number and the offset within the segment. The user therefore specifies each address by two quantities: a segment number and an offset. (Contrast this scheme with the paging scheme, in which the user specifies only a single address, which is partitioned by the hardware into a page number and an offset, all invisible to the programmer.)

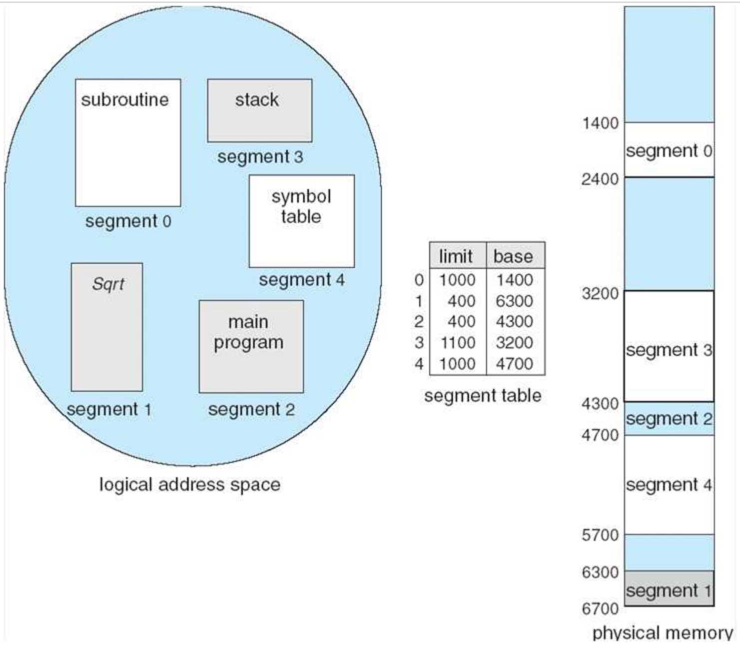


Figure 4.9: Example of segmentation.

A segment table is used to map segments into main memory. Each entry in the segment table has a segment base and a segment limit. The segment base contains the starting physical address where the segment resides in memory, whereas the segment limit specifies the length of the segment.

[19] For 200 ns, or 0.2 µs, the delay is (8.2 – 0.2) or 8 µs.

∴ For 100 µs, the delay is $\left(\frac{8}{0.2} \times 100\right)$ or 40 µs.

∴ The delay factor is 40.

Comparison of *Paging* and *Segmentation*

Consideration	Paging	Segmentation
Need the programmer be aware that this technique is being used?	No	Yes
How many linear address spaces are there?	1	Many
Can the total address space exceed the size of physical memory?	Yes	Yes
Can procedures and data be distinguished and separately protected?	No	Yes
Can tables whose size fluctuates be accommodated easily?	No	Yes
Is sharing of procedures between users facilitated?	No	Yes
Why was this technique invented?	To get a large linear address space without having to buy more physical memory	To allow programs and data to be broken up into logically independent address spaces and to aid sharing and protection

4.10 Page Replacement Algorithms

While it would be possible to pick a random page to evict at each page fault, system performance is much better if a page that is not heavily used is chosen. If a heavily used page is removed, it will probably have to be brought back in quickly, resulting in extra overhead. Much work has been done on the subject of page replacement algorithms, both theoretical and experimental. Below we will describe some of the most important algorithms.

The Optimal Page Replacement Algorithm (OPT / MIN)

- The page that will not be used for the longest period of time is to be replaced.
- For example, if one page will not be used for 8 million instructions and another page will not be used for 6 million instructions, we should choose the former page to replace.
- **Problem:**

  The algorithm is unrealizable. At the time of the page fault, the operating system has no way of knowing when each of the pages will be referenced next.

  A possible solution to this problem:

  By running a program on a simulator and keeping track of all page references, it is possible to implement optimal page replacement on the *second* run by using the page reference information collected during the *first* run.

  The problem with this solution:

  This log of page references refers only to the *one* program just measured and then with *only one specific input*. The page replacement algorithm derived from it is thus specific to that one program and input data.
- **A use of this algorithm:**

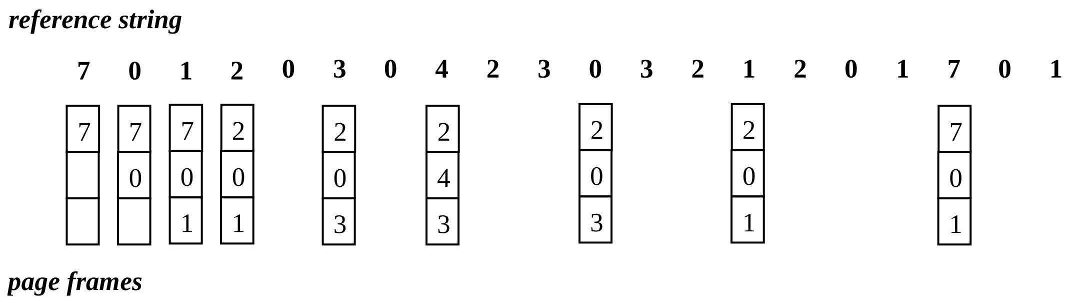
  Although this algorithm is of no use in practical systems, it is useful for evaluating other practical page replacement algorithms.

Example:

Let, we have only three frames; and the page reference string is:

7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

Then, the page frames after each reference string is shown below:



Not Recently Used (NRU)

- In order to allow the operating system to collect useful statistics about which pages are being used and which ones are not, most computers with virtual memory have two status bits associated with each page. *R* is set whenever the page is referenced (read or written). *M* is set when the page is written to (i.e., modified). The bits are contained in each page table entry.
- When a process is started up, both page bits for all its pages are set to 0 by the operating system. When a page fault occurs, the *R* bit is set. When the content of a page is modified, the *M* bit is set. Periodically (e.g., on each clock interrupt), the *R* bit is cleared, to distinguish pages that have not been referenced *recently* from those that have been.
- When a page fault occurs, the operating system inspects all the pages and divides them into four categories based on the current values of their *R* and *M* bits:

  Class 0: not referenced, not modified.

  Class 1: not referenced, modified.

  Class 2: referenced, not modified.

  Class 3: referenced, modified.
- The NRU algorithm removes a page at random from the *lowest* numbered *nonempty* class.
- Implicit in this algorithm is that it is better to remove a modified page that has not been referenced in at least one clock tick (typically 20 msec) than a clean page that is in heavy use.
- **Advantages:**
  1. Easy to understand
  2. Moderately efficient to implement
  3. Gives a performance that, while certainly not optimal, may be adequate.

First-In, First-Out (FIFO)

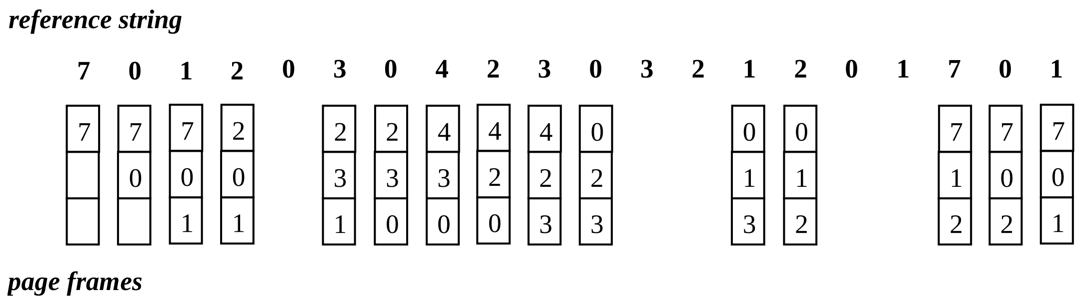
- The operating system maintains a list of all pages currently in memory, with the page at the head of the list the oldest one and the page at the tail the most recent arrival. On a page fault, the page at the head is removed and the new page added to the tail of the list.
- **Advantage:** Low overhead and easy to implement.
- **Disadvantage:** The older pages might be heavily used pages, and removing them might cause more page faults.

Example:

Let, we have only three frames; and the page reference string is:

7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

Then, the page frames after each reference string is shown below:



Second Chance

- A simple modification to FIFO that avoids the problem of throwing out a heavily used page is to inspect the *R* bit of the oldest page. If it is 0, the page is both old and unused, so it is replaced immediately. If the *R* bit is 1, the bit is cleared, the page is put onto the end of the list of pages, and its load time is updated as though it had just arrived in memory. Then the search continues.
- Suppose that a page fault occurs at time 20. The oldest page is *A*, which arrived at time 0, when the process started. If *A* has the *R* bit cleared, it is evicted from memory, either by being written to the disk (if it is dirty), or just abandoned (if it is dean). On the other hand, if the *R* bit is set, *A* is put onto the end of the list and its "load time" is reset to the current time (20). The *R* bit is also cleared. The search for a suitable page continues with *B*.

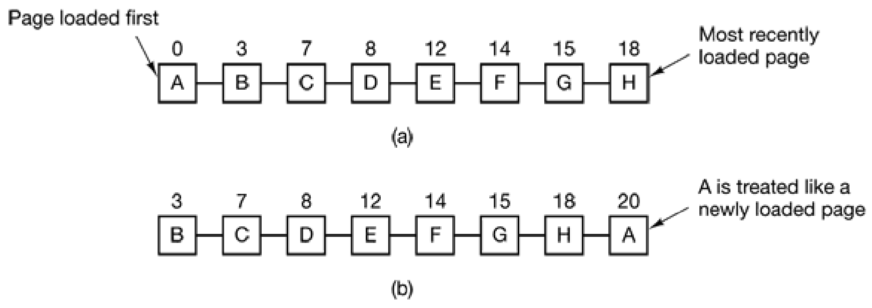


Figure: Operation of second chance.
(a) Pages sorted in FIFO order.
(b) Page list if a page fault occurs at time 20 and *A* has its *R* bit set.
The numbers above the pages are their loading times.

- What second chance is doing is looking for an old page that has not been referenced in the previous clock interval.
- If all the pages have been referenced, second chance degenerates into pure FIFO.

Second Chance (Clock) / Clock Page Replacement Algorithm

The problem with Second Chance algorithm

The second chance algorithm is unnecessarily inefficient because it is constantly moving pages around on its list.

How the clock algorithm solves this

A better approach is to keep all the page frames on a circular linked list in the form of a clock, and a hand points to the *oldest* page.

Operation of this algorithm

When a page fault occurs, the page being pointed to by the hand is inspected.

a. If its *R* bit is 0,
   1. The page is evicted.

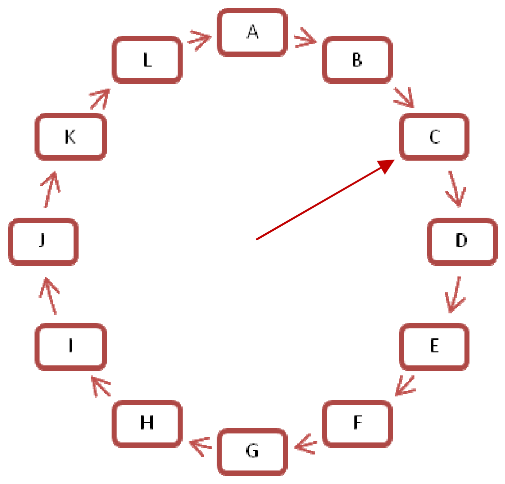


Figure: The clock page replacement algorithm.

    2. The new page is inserted into the clock in its place.
    3. The hand is advanced one position.

b. If *R* is 1,

    1. *R* is cleared.
    2. The hand is advanced to the next page.
    3. This process is repeated until a page is found with *R* = 0.

Least Recently Used (LRU)

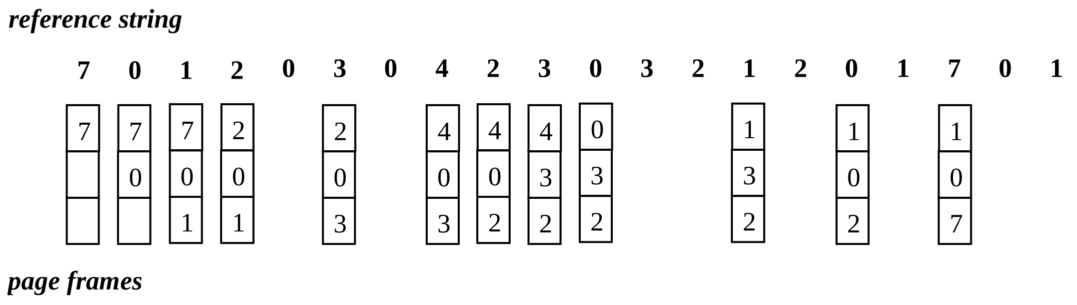
- ➢ A good approximation to the optimal algorithm is based on the observation that pages that have been heavily used in the last few instructions will probably be heavily used again in the next few. Conversely, pages that have not been used for ages will probably remain unused for a long time.
- ➢ This idea suggests a realizable algorithm: when a page fault occurs, throw out the page that has been unused for the longest time. This strategy is called LRU (Least Recently Used) paging.

Example:

Let, we have only three frames; and the page reference string is:

7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

Then, the page frames after each reference string is shown below:



Implementing LRU Using Hardware

Method 1

- ➢ A 64-bit hardware counter, *C*, is automatically incremented after each instruction.
- ➢ Each page table entry must also have a field large enough to contain the value of *C*.
- ➢ After each memory reference, the current value of *C* is stored in the page table entry for the page just referenced.
- ➢ When a page fault occurs, the operating system examines all the counters in the page table to find the lowest one. That page is the least recently used.

Method 2

- ➢ For a machine with *n* page frames, a matrix of $n \times n$ bits, initially all zero, is used.
- ➢ Whenever page frame *k* is referenced, the hardware first sets all the bits of row *k* to 1, and then sets all the bits of column *k* to 0.
- ➢ At any instant, the *row* whose binary value is *lowest* is the least recently used; the row whose value is next lowest is next least recently used; and so forth.

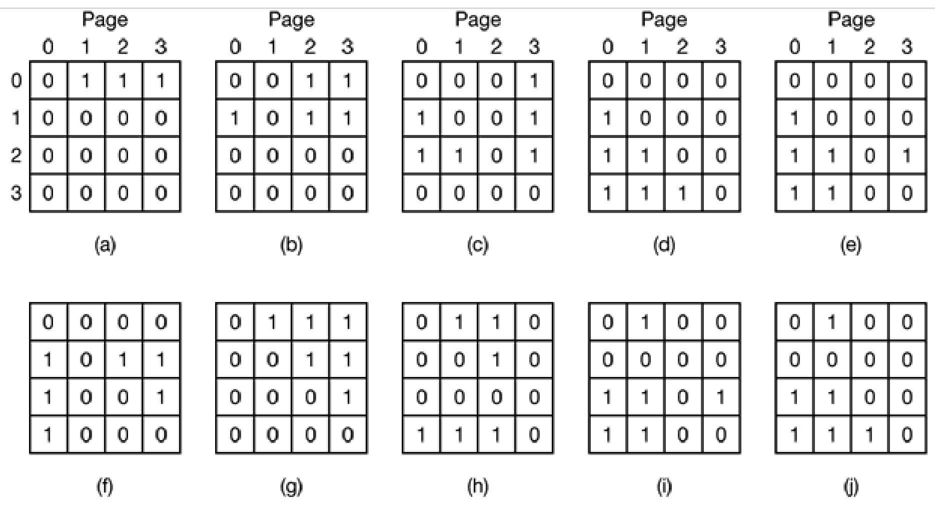


Figure: LRU using a matrix when pages are referenced in the order 0, 1, 2, 3, 2, 1, 0, 3, 2, 3.

Simulating LRU in Software

Method 1: Not Frequently Used (NFU) / Least Frequently Used (LFU) Algorithm

- ➢ A software counter is associated with each page, initially zero.
- ➢ At each clock interrupt, the operating system scans all the pages in memory. For each page, the *R* bit, which is 0 or 1, is added to the counter. In effect, the counters are an attempt to keep track of how often each page has been referenced.
- ➢ When a page fault occurs, the page with the lowest counter is chosen for replacement.

Method 2: Aging Algorithm

The Problem with NFU Algorithm:

The main problem with NFU is that it never forgets anything. For example, in a multipass compiler pages that were heavily used during pass 1 may still have a high count well into later passes. In fact, if pass 1 happens to have the longest execution time of all the passes, the pages containing the code for subsequent passes may always have lower counts than the pass 1 pages. Consequently, the operating system will remove useful pages instead of pages no longer in use.

How the Aging Algorithm Solves it:

Fortunately, a small modification to NFU makes it able to simulate LRU quite well. The modification has two parts:

1. The counters are each shifted right 1 bit before the *R* bit is added in.
2. The *R* bit is added to the leftmost, rather than the rightmost bit.

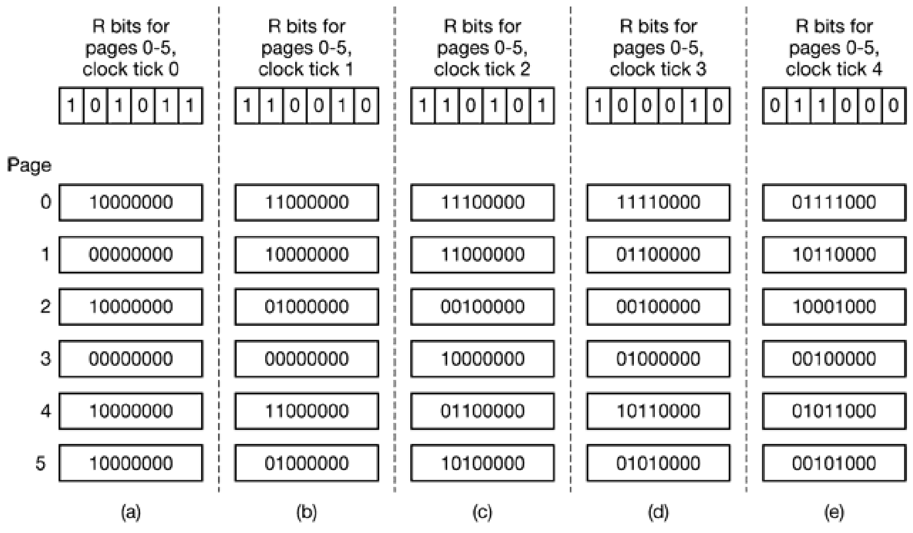


Figure: The aging algorithm simulates LRU in software.
Shown are six pages for five clock ticks.
The five clock ticks are represented by (a) to (e).

4.10 The Working Set (WS) Model

Most processes exhibit a ***locality of reference***, meaning that during any phase of execution, the process references only a relatively small fraction of its pages. Each pass of a multipass compiler, for example, references only a fraction of all the pages.

The set of pages that a process is currently using is called its ***working set***.

If the entire working set is in memory, the process will run without causing many faults until it moves into another execution phase (e.g., the next pass of the compiler).

If the available memory is too small to hold the entire working set, the process will cause many page faults and run slowly since executing an instruction takes a few nanoseconds and reading in a page from the disk typically takes 10 milliseconds.

A program causing page faults every few instructions is said to be ***thrashing***.

In a multiprogramming system, processes are frequently moved to disk (i.e., all their pages are removed from memory) to let other processes have a turn at the CPU. The question arises of *what to do when a process is brought back in again*. Technically, nothing need be done. The process will just cause page faults until its working set has been loaded. The problem is that having 20, 100, or even 1000 page faults every time a process is loaded is slow, and it also wastes considerable CPU time, since it takes the operating system a few milliseconds of CPU time to process a page fault.

Therefore, many paging systems try to keep track of each process' working set and make sure that it is in memory before letting the process run. This approach is called the ***working set model***. It is designed to greatly reduce the page fault rate. Loading the pages *before* letting processes run is also called ***prepaging***. Note that the working set changes over time.

To implement the working set model, it is necessary for the operating system to keep track of which pages are in the working set. Having this information also immediately leads to a possible page replacement algorithm: when a page fault occurs, find a page not in the working set and evict it.

This model uses a parameter, Δ, to define the working-set window. The idea is to examine the most recent Δ page references. The set of pages in the most recent Δ page references is the working set. If a page is in active use, it will be in the working set. If it is no longer being used, it will drop from the working set Δ time units after its last reference. Thus, the working set is an approximation of the program's locality.

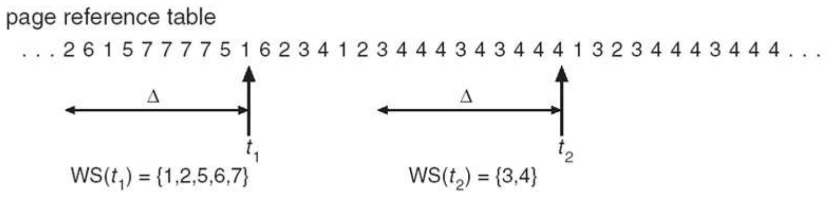


Figure: Working Set Model.

For example, given the sequence of memory references shown in the above figure, if Δ = 10 memory references, then the working set at time t_1 is {1, 2, 5, 6, 7}. By time t_2, the working set has changed to {3, 4}.

4.11 Page Fault Frequency (PFF) – A Strategy to Prevent Thrashing

When the page fault rate is too high, we know that the process needs more frames. Conversely, if the page fault rate is too low, then the process may have too many frames. We can establish upper and lower bounds on the desired page fault rate. If the actual page fault rate exceeds the upper limit, we allocate the process another frame; if the page fault rate falls below the lower limit, we remove a frame from the process. Thus, we can directly measure and control the page fault rate to prevent thrashing.

As with the working set strategy, we may have to suspend a process. If the page fault rate increases and no free frames are available, we must select some process and suspend it. The freed frames are then distributed to processes with high page fault rates.

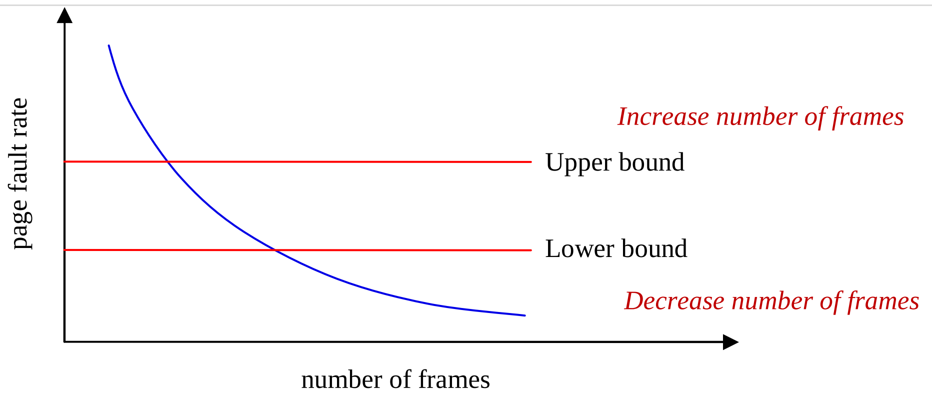


Figure: Page Fault Frequency.

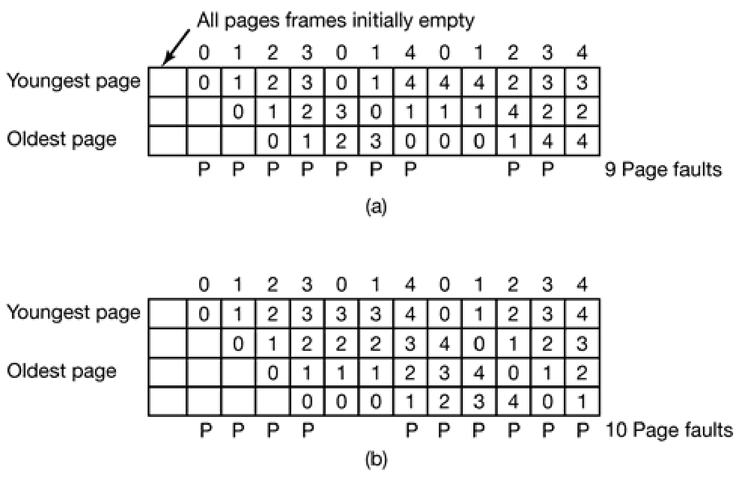
4.12 Belady's Anomaly

Intuitively, it might seem that the more page frames the memory has, the fewer page faults a program will get. Surprisingly enough, this is not always the case. Belady et al. discovered a counterexample, in which FIFO caused more page faults with four page frames than with three. This strange situation has become known as *Belady's anomaly*.

It is illustrated in the following figure for a program with five virtual pages, numbered from 0 to 4. The pages are referenced in the order

0 1 2 3 0 1 4 0 1 2 3 4

In *figure* 4.12(*a*), we see how with three page frames a total of nine page faults are caused. In *figure* 4.12(*b*), we get ten page faults with four page frames.



		0	1	2	3	0	1	4	0	1	2	3	4	
Youngest page		0	1	2	3	0	1	4	4	4	2	3	3	
			0	1	2	3	0	1	1	1	4	2	2	
Oldest page				0	1	2	3	0	0	0	1	4	4	
		P	P	P	P	P	P	P			P	P		9 Page faults

(a)

		0	1	2	3	0	1	4	0	1	2	3	4	
Youngest page		0	1	2	3	3	3	4	0	1	2	3	4	
			0	1	2	2	2	3	4	0	1	2	3	
Oldest page				0	1	1	1	2	3	4	0	1	2	
					0	0	0	1	2	3	4	0	1	
		P	P	P	P			P	P	P	P	P	P	10 Page faults

(b)

Figure 4.12: Belady's anomaly.
(a) FIFO with three page frames.
(b) FIFO with four page frames.
The *P*'s show which page references cause page faults.

4.13 Frame Allocation

How do we allocate the fixed amount of free memory among the various processes? If we have 93 free frames and two processes, how many frames does each process get?

Frame Allocation Algorithms

Equal Allocation

The easiest way to split *m* frames among *n* processes is to give everyone an equal share, *m* / *n* frames. For instance, if there are 93 frames and five processes each process will get 18 frames. The leftover three frames can be used as a free-frame buffer pool. This scheme is called *equal allocation*.

Disadvantage:

Various processes will need differing amounts of memory. So, frames might be wasted.

Proportional Allocation

Available memory is allocated to each process according to its size. Let the size of the virtual memory for process p_i be s_i, and define

$$S = \sum s_i$$

Then, if the total number of available frames is m, we allocate a_i frames to process p_i; where a_i is approximately

$$a_i = \left\lfloor \frac{s_i}{S} \times m \right\rfloor$$

For proportional allocation, we would split 62 frames between two processes, one of 10 pages and one of 127 pages, by allocating 4 frames and 57 frames, respectively, since

$$\frac{10}{137} \times 62 \approx 4, \text{ and } \frac{127}{137} \times 62 \approx 57$$

Notice that, with either equal or proportional allocation, a high-priority process is treated the same as a low-priority process. By its definition, however, we may want to give the high-priority process more memory to speed its execution, to the detriment of low-priority processes. One solution is to use a proportional allocation scheme wherein the ratio of frames depends not on the relative sizes of processes but rather on the priorities of processes or on a combination of size and priority.

Global Versus Local Allocation

Another important factor in the way frames are allocated to the various processes is page replacement. With multiple processes competing for frames, we can classify page-replacement algorithms into two broad categories: *global replacement* and *local replacement*.

Global replacement allows a process to select a replacement frame from the set of all frames, even if that frame is currently allocated to some other process; that is, one process can take a frame from another. *Local replacement* requires that each process select from only its own set of allocated frames.

For example, consider an allocation scheme where we allow high-priority processes to select frames from low-priority processes for replacement. A process can select a replacement from among its own frames or the frames of any lower-priority process. This approach allows a high-priority process to increase its frame allocation at the expense of a low-priority process.

With a local replacement strategy, the number of frames allocated to a process does not change. With global replacement, a process may happen to select only frames allocated to other processes, thus increasing the number of frames allocated to it (assuming that other processes do not choose its frames for replacement).

4.14 Locking / Pinning Pages in Memory (I/O Interlock)

We must be sure the following sequence of events does not occur: A process issues an I/O request and is put in a queue for that I/O device. Meanwhile, the CPU is given to other processes. These processes cause page faults; and one of them, using a global replacement algorithm, replaces the page containing the memory buffer for the waiting process. The pages are paged out. Some time later, when the I/O request advances to the head of the device queue, the I/O occurs to the specified address. However, this frame is now being used for a different page belonging to another process.

There are two common solutions to this problem. One solution is never to execute I/O to user memory. Instead, data are always copied between system memory and user memory. I/O takes place only between system memory and the I/O device. To write a block on tape, we first copy the block to system memory and then write it to tape. This extra copying may result in unacceptably high overhead.

Another solution is to allow pages to be locked into memory. Here, a lock bit is associated with every frame. If the frame is locked, it cannot be selected for replacement. Under this approach, to write a block on tape, we lock into memory the pages containing the block. The system can then continue as usual. Locked pages cannot be replaced. When the I/O is complete, the pages are unlocked.

Chapter 5
Input/Output

Roadmap and Concepts in Brief

- In this chapter, we'll learn how operating systems manage I/O devices.
- First of all, how can the processor give commands and data to a controller to accomplish an I/O transfer?

  By writing bits to device registers. Device registers can be accessed in any of the following ways:

  1. Port I/O
  2. Memory-Mapped I/O
  3. Hybrid of Port and Memory-Mapped I/O
  4. DMA (Direct Memory Access)
- How I/O can be performed?
  1. Programmed I/O
  2. Interrupt-Driven I/O
  3. I/O using DMA
- Describe the disk arm scheduling algorithms.
  1. **First-Come First-Serve (FCFS)**

     Requests are served in first-come first-serve manner.
  2. **Shortest Seek First (SSF) / Shortest Seek Time First (SSTF)**

     Selects the request with the minimum seek time from the current head position.
  3. **SCAN Scheduling / Elevator Algorithm**

     The disk arm starts at one end of the disk and moves toward the other end, servicing requests as it reaches each cylinder, until it gets to the other end of the disk. At the other end, the direction of head movement is reversed, and servicing continues. The head continuously scans back and forth across the disk.
     4. **Circular SCAN (C-SCAN) / Modified Elevator**

        Like SCAN, C-SCAN moves the head from one end of the disk to the other, servicing requests along the way. When the head reaches the other end, however, it immediately returns to the beginning of the disk, without servicing any requests on the return trip.
     5. **LOOK and C-LOOK**

        The arm goes only as far as the final request in each direction. Then, it reverses direction immediately, without going all the way to the end of the disk.

Theory

5.1 Categories of I/O Devices

- **Character-Stream or Block:**

  A character-stream device transfers bytes one by one, whereas a block device transfers a block of bytes as a unit.

  Disks are the most common block devices. Printers, network interfaces, mice and most other devices that are not disk-like can be seen as character devices.

- **Sequential or Random-Access:**

  A sequential device transfers data in a fixed order determined by the device, whereas the user of a random-access device can instruct the device to seek to any of the available data storage locations.

  Tape drives are sequential devices, whereas hard disk drives are random-access devices.

- **Sharable or Dedicated:**

  A sharable device can be used concurrently by several processes or threads; a dedicated device cannot.

  CD/DVD recorders and printers are dedicated devices, whereas hard disk drives are sharable devices.

- **Synchronous or Asynchronous:**

  A synchronous device performs data transfers with predictable response times. An asynchronous device exhibits irregular or unpredictable response times.

5.2 Device Controller

A device controller is a collection of electronics that can operate a port, a bus, or a device.

A serial-port controller is a simple device controller. It is a single chip (or portion of a chip) in the computer that controls the signals on the wires of a serial port. By contrast, a SCSI bus controller is not simple. Because the SCSI protocol is complex, the SCSI bus controller is often implemented as a separate circuit board (or a **host adapter**) that plugs into the computer. It typically contains a processor, microcode, and some private memory to enable it to process the SCSI protocol messages. Some devices have their own built-in controllers. If you look at a disk drive, you will see a circuit board attached to one side. This board is the disk controller. It implements the disk side of the protocol for some kind of connection – SCSI or ATA, for instance. It has microcode and a processor to do many tasks, such as bad-sector mapping, prefetching, buffering and caching.

How a Controller Works

The interface between the controller and the device is often a very low-level interface. A disk, for example, might be formatted with 256 sectors of 512 bytes per track. What actually comes off the drive, however, is a serial bit stream, starting with a preamble, then the 512 bytes in a sector, and finally a checksum (ECC). The controller's job is to convert the serial bit stream into a block of bytes and perform any error correction necessary. The block of bytes is typically first assembled, bit by bit, in a buffer inside the controller. After its checksum has been verified and the block declared to be error free, it can then be copied to main memory. The operating system initializes the controller with a few parameters, such as which blocks are to be read, where in the memory they should be stored etc.

5.3 How Can the Processor Give Commands and Data to a Controller to Accomplish an I/O Transfer

Each controller has a few control and status registers that are used for communicating with the CPU. By writing into these registers, the operating system can command the device to deliver data, accept data, switch itself on or off, or otherwise perform some action. By reading from these registers, the operating system can learn what the device's state is, whether it is prepared to accept a new command, and so on.

In addition to the control registers, many devices have a data buffer that the operating system can read and write. For example, a common way for computers to display pixels on the screen is to have a video RAM, which is basically just a data buffer, available for programs or the operating system to write into.

How the Device and Control Registers Can Be Accessed

Port I/O

Each control register is assigned an I/O port number, an 8- or 16-bit integer. Using special I/O instructions such as `IN`, `OUT`, the CPU can read in or write to control registers.

In this scheme, the address spaces for memory and I/O are different, as shown in *figure* 5.3(*a*). The instructions `IN R0,4` and `MOV R0,4` are completely different in this design. The former reads the contents of I/O port 4 and puts it in R0 whereas the latter reads the contents of memory word 4 and puts it in R0. The 4s in these examples thus refer to different and unrelated address spaces.

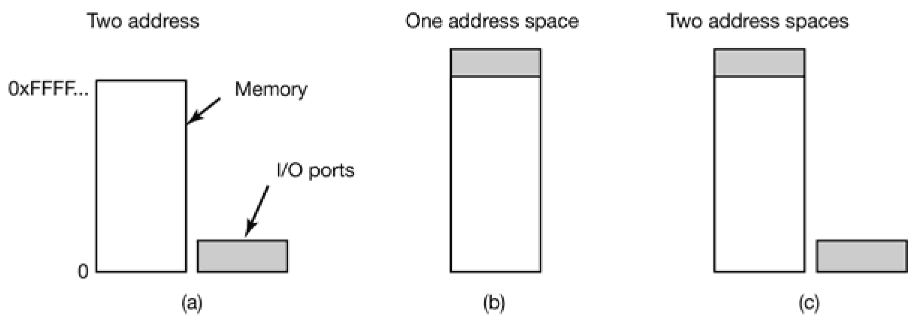


Figure 5.3: (a) Separate I/O and memory space. (b) Memory mapped I/O. (c) Hybrid.

Memory-Mapped I/O

All the control registers are mapped into the memory space, as shown in *figure* 5.3(*b*). Each control register is assigned a unique memory address to which no memory is assigned. Usually, the assigned addresses are at the top of the address space.

A hybrid scheme, with memory-mapped I/O data buffers and separate I/O ports for the control registers is shown in *figure* 5.3(*c*). The Pentium uses this architecture, with addresses 640K to 1M being reserved for device data buffers in IBM PC compatibles, in addition to I/O ports 0 through 64K.

How do these schemes work?

In all cases, when the CPU wants to read a word, either from memory or from an I/O port, it puts the address it needs on the bus' address lines and then asserts a READ signal on a bus' control line. A second signal line is used to tell whether I/O space or memory space is needed. If it is memory space, the memory responds to the request. If it is I/O space, the I/O device responds to the request. If there is only memory space [as in *figure* 5.3(*b*)], every memory module and every I/O device compares the address lines to the range of addresses that it services. It the address falls in its range, it responds to the request. Since no address is ever assigned to both memory and an I/O device, there is no ambiguity and no conflict.

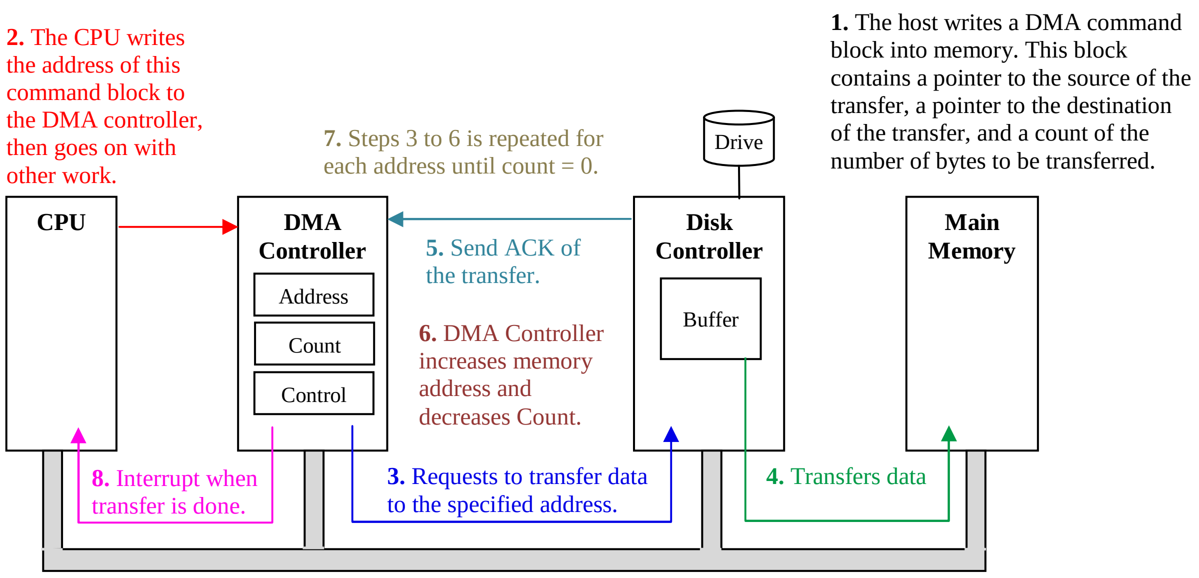
DMA (Direct Memory Access)

The CPU can request data from an I/O controller one byte at a time but doing so wastes the CPU's time, so a different scheme, called *DMA* **(***Direct Memory Access***)** is often used.

A DMA controller contains several registers that can be written and read by the CPU. These include a memory address register, a byte count register, and one or more control registers. The control registers specify the I/O port to use, the direction of the transfer (reading from the I/O device or writing to the I/O device), the transfer unit (byte at a time or word at a time), and the number of bytes to transfer in one burst.

Sometimes this controller is integrated into disk controllers and other controllers, but such a design requires a separate DMA controller for each device. More commonly, a single DMA controller is available (e.g., on the motherboard) for regulating transfers to multiple devices, often concurrently.

How DMA Works



2. The CPU writes the address of this command block to the DMA controller, then goes on with other work.
7. Steps 3 to 6 is repeated for each address until count = 0.
Drive
1. The host writes a DMA command block into memory. This block contains a pointer to the source of the transfer, a pointer to the destination of the transfer, and a count of the number of bytes to be transferred.
CPU
DMA Controller
Address
Count
Control
5. Send ACK of the transfer.
6. DMA Controller increases memory address and decreases Count.
Disk Controller
Buffer
Main Memory
8. Interrupt when transfer is done.
3. Requests to transfer data to the specified address.
4. Transfers data

DMA Operation Modes

1. **Word-at-a-time Mode (*Cycle Stealing*):** The DMA controller requests for the transfer of one word at a time.
2. **Block Mode (*Burst Mode*):** The DMA controller tells the device to acquire the bus, issue a series of transfers, and then release the bus.

5.4 **Ways of Performing I/O**

Programmed I/O

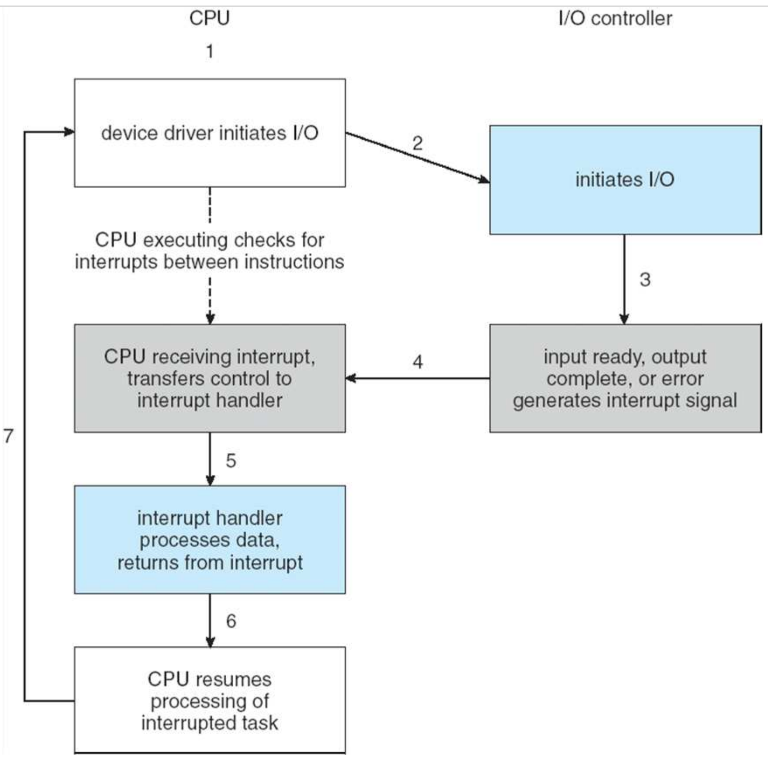
The CPU Continuously polls the device to see if it is ready to accept a request. This behavior is often called *polling* or *busy waiting*.

Advantage: Simple.
Disadvantage: The CPU is tied up full time until all the I/O is done.

Interrupt-Driven I/O

See *figure* 5.4.

Disadvantage: Interrupts take time, so this scheme wastes a certain amount of CPU time.



CPU
I/O controller
1
device driver initiates I/O
2
initiates I/O
CPU executing checks for interrupts between instructions
3
CPU receiving interrupt, transfers control to interrupt handler
4
input ready, output complete, or error generates interrupt signal
7
5
interrupt handler processes data, returns from interrupt
6
CPU resumes processing of interrupted task

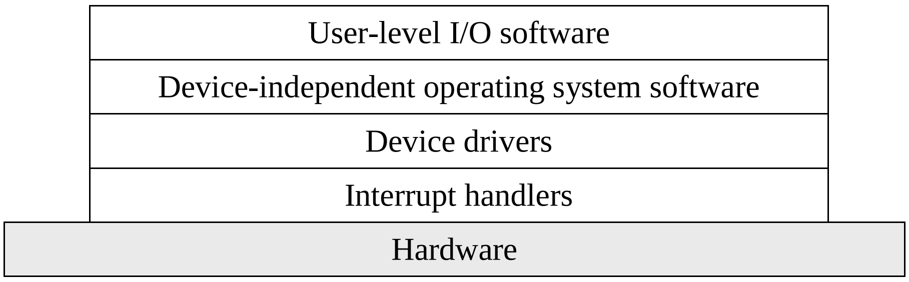
Figure 5.4: Interrupt-driven I/O cycle.

I/O using DMA

In essence, DMA is programmed I/O, only with the DMA controller doing all the work, instead of the main CPU.

Advantage: The number of interrupts is reduced.
Disadvantage: the DMA controller is usually much slower than the main CPU. If the DMA controller is not capable of driving the device at full speed, or the CPU usually has nothing to do anyway while waiting for the DMA interrupt, then interrupt-driven I/O or even programmed I/O may be better.

5.5 I/O Software Layers



5.6 How Interrupts are Handled

Below is a series of steps that must be performed in software after the hardware interrupt has completed. It should be noted that the details are very system dependent, so some of the steps listed below may not be needed on a particular machine and steps not listed may be required. Also, the steps that do occur may be in a different order on some machines.

1. Save any registers (including the PSW) that have not already been saved by the interrupt hardware.
2. Set up a context for the interrupt service procedure. Doing this may involve setting up the TLB, MMU and a page table.
3. Set up a stack for the interrupt service procedure.
4. Acknowledge the interrupt controller. If there is no centralized interrupt controller, re-enable interrupts.
5. Copy the registers from where they were saved (possibly some stack) to the process table.
6. Run the interrupt service procedure. It will extract information from the interrupting device controller’s registers.
7. Choose which process to run next. If the interrupt has caused some high-priority process that was blocked to become ready, it may be chosen to run now.
8. Set up the MMU context for the process to run next. Some TLB set up may also be needed.
9. Load the new process’ registers, including its PSW.
10. Start running the new process.

On some machines, the TLB and CPU cache may also have to be managed when switching between user and kernel modes, which takes additional machine cycles.

5.7 Device Drivers

Each I/O device attached to a computer needs some device-specific code for controlling it. This code, called the *device driver*, is generally written by the device’s manufacturer and delivered along with the device.

Each device driver normally handles one device type, or at most, one class of closely related devices. For example, a SCSI disk driver can usually handle multiple SCSI disks of different sizes and different speeds, and perhaps a SCSI CD-ROM as well.

In some systems, the operating system is a single binary program that contains all of the drivers that it will need compiled into it. In newer systems, they are loaded dynamically when needed.

Functions of a Device Driver

1. Translate request through the device-independent standard interface (open, close, read, write) into appropriate sequence of commands (register manipulations) for the particular hardware.
2. Initialize the hardware and shut it down cleanly.
3. Might also need to manage power requirements of the hardware and log events.

5.8 Device-Independent I/O Software

The basic function of the device-independent software is to perform the I/O functions that are common to all devices and to provide a uniform interface to the user-level software.

The following functions are typically done in the device-independent software:

Uniform Interfacing for Device Drivers

If each device driver has a different interface to the operating system, then the driver functions available for the system to call differ from driver to driver. The kernel functions that the driver needs might also differ from driver to driver. Taken together, it means that interfacing each new driver requires a lot of new programming effort.

Buffering

How a Non-Buffered System Works

Consider a user process that wants to read data from a modem. The process does a `read` system call and block waiting for one character. Each arriving character causes an interrupt. The interrupt service procedure hands the character to the user process and unblocks it. After pulling the character somewhere, the process reads another character and blocks again.

Problem With this System

The user process has to be started up for every incoming character Allowing a process to run many times for short runs is inefficient, so this design is not a good one.

Buffering in User Space

The user process provides an *n*-character buffer in user space and does a read of *n* characters. The interrupt service procedure puts incoming characters in this buffer until it fills up. Then it wakes up the user process.

Problem With this System

What happens if the buffer is paged out when a character arrives? The buffer could be locked in memory, but if many processes start locking pages in memory, the pool of available pages will shrink and performance will degrade.

Buffering in Kernel Space

A buffer inside the kernel is created and the interrupt handler puts the characters there. When this buffer is full, the page with the user buffer is brought in, if needed, and the buffer copied there in one operation.

Problem With this System

What happens to characters that arrive while the page with the user buffer is being brought in from the disk? Since the buffer is full, there is no place to put them.

Double Buffering

A way out is to have a second kernel buffer. After the first buffer fills up, but before it has been emptied, the second one is used. When the second buffer fills up, it is available to be copied to the user (assuming the user has asked for it). While the second buffer is being copied to user space, the first one can be used for new characters.

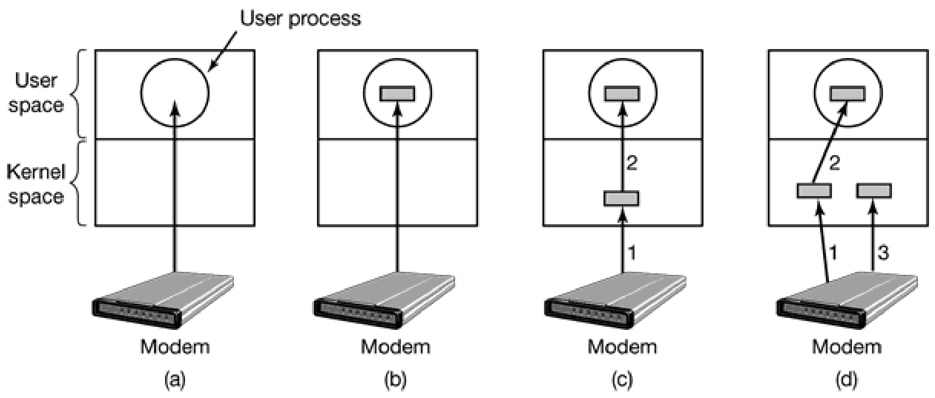


Figure 5.8: (a) Unbuffered input. (b) Buffering in user space.
(c) Buffering in the kernel followed by copying to user space.
(d) Double buffering in the kernel.

Why Buffering is Needed

1. **To cope with a speed mismatch between the producer and consumer of a data stream.**

   Suppose, for example, that a file is being received via modem for storage on the hard disk. The modem is about a thousand times slower than the hard disk. So a buffer is created in main memory to accumulate the bytes received from the modem. When an entire buffer of data has arrived, the buffer can be written to disk in a single operation.

2. **To adapt between devices that have different data-transfer sizes.**

   Such disparities are especially common in computer networking, where buffers are used widely for fragmentation and reassembly of messages. At the sending side, a large message is fragmented into small network packets. The packets are sent over the network, and the receiving side places them in a reassembly buffer to form an image of the source data.

5.9 User-Space I/O Software

Although most of the I/O software is within the operating system, a small portion of it consists of libraries linked together with user programs, and even whole programs running outside the kernel. System calls, including the I/O system calls, are normally made by library procedures. When a C program contains the call

```
count = write(fd, buffer, nbytes);
```

the library procedure *write* will be linked with the program and contained in the binary program present in memory at run time. The collection of all these library procedures is clearly part of the I/O system.

5.10 Summary of the I/O System

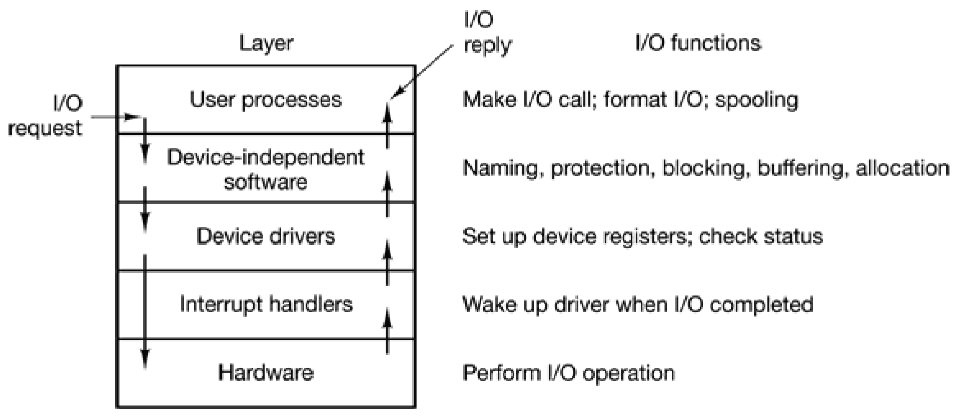

Layer
I/O reply
I/O functions
I/O request
User processes
Device-independent software
Device drivers
Interrupt handlers
Hardware
Make I/O call; format I/O; spooling
Naming, protection, blocking, buffering, allocation
Set up device registers; check status
Wake up driver when I/O completed
Perform I/O operation

Figure 5.10: Layers of the I/O system and the main functions of each layer.

5.11 Magnetic Disk Structure

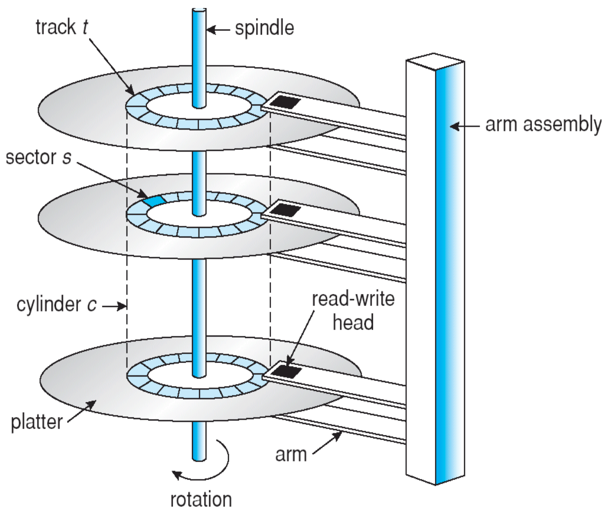


Figure 5.11(a): Moving-head magnetic disk mechanism.

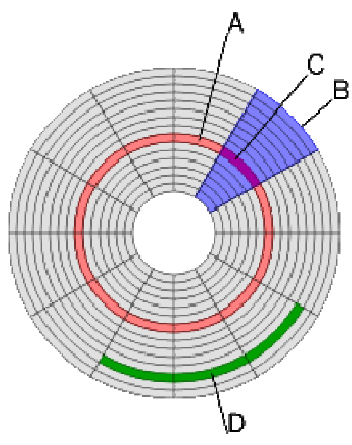


Figure 5.11(b): Disk Structure: (A) Track (B) Geometrical Sector (C) Track sector (D) Block / Cluster

5.12 Disk Geometry

On older disks, the number of sectors per track was the same for all cylinders. Modern disks are divided into zones with more sectors on the outer zones than the inner ones. *Figure* 5.12.1(*a*) illustrates a tiny disk with two zones. The outer zone has 32 sectors per track: the inner one has 16 sectors per track.

To hide the details of how many sectors each track has, most modern disks have a virtual geometry that is presented to the operating system. The software is instructed to act as though there are *x* cylinders, *y* heads, and *z* sectors per track. The controller then remaps a request for (*x*, *y*, *z*) onto the real cylinder, head, and sector. A possible virtual geometry for the physical disk of *figure* 5.12.1(*a*) is shown in *figure* 5.12.1(*b*). In both cases the disk has 192 sectors, only the published arrangement is different than the real one.

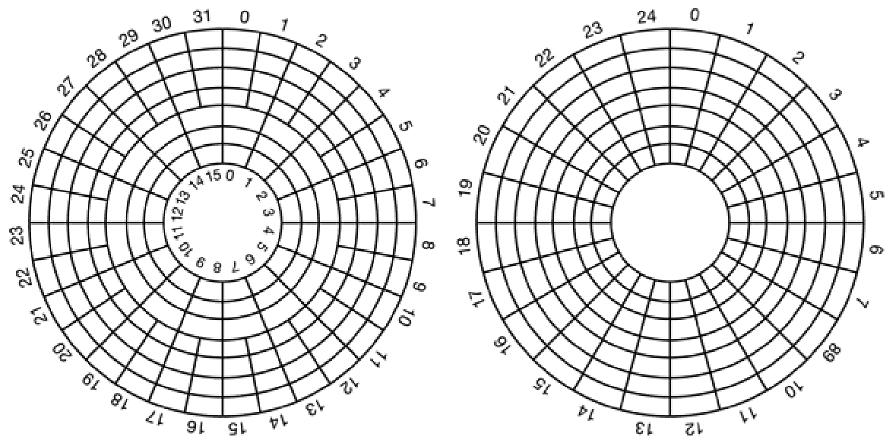


Figure 5.12.1: (a) Physical geometry of a disk with two zones. (b) A possible virtual geometry for this disk.

Cylinder Skew

The position of sector 0 on each track is offset from the previous track when the low-level format is laid down. This offset, called *cylinder skew*, is done to improve performance. The idea is to allow the disk to read multiple tracks in one continuous operation without losing data.

The nature of the problem can be seen by looking at *figure* 5.12.1(*a*). Suppose that a request needs 18 sectors starting at sector 0 on the innermost track. Reading the first 16 sectors takes one disk rotation, but a seek is needed to move outward one track to get the 17th sector. By the time the head has moved one track, sector 0 has rotated past the head so an entire rotation is needed until it comes by again. That problem is eliminated by offsetting the sectors as shown in *figure* 5.12.2.

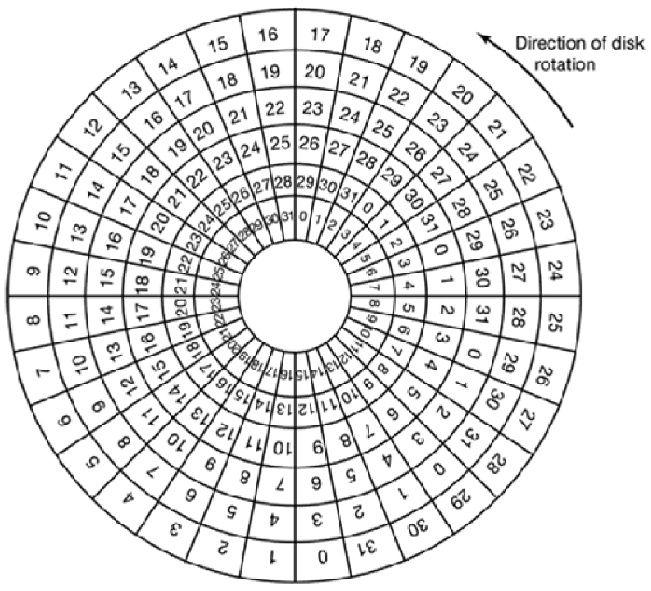


Figure 5.12.2: An illustration of cylinder skew.

5.13 Disk Formatting

Low-Level / Physical Formatting

A new magnetic disk is a blank slate: it is just a platter of a magnetic recording material. Before a disk can store data, it must be divided into sectors that the disk controller can read and write. This process is called *low-level formatting* or *physical formatting*.

The format consists of a series of concentric tracks, each containing some number of sectors, with short gaps between the sectors. The format of a sector is shown in *figure* 5.13.

Preamble	Data	ECC

Figure 5.13: A disk sector.

The preamble starts with a certain bit pattern that allows the hardware to recognize the start of the sector. It also contains the cylinder and sector numbers and some other information.

The size of the data portion is determined by the low-level formatting program. Most disks use 512-byte sectors.

The ECC field contains redundant information that can be used to recover from read errors. The size and content of this field varies from manufacturer to manufacturer, depending on how much disk space the designer is willing to give up for higher reliability and how complex an ECC code the controller can handle. A 16-byte ECC field is not unusual. Furthermore, all hard disks have some number of spare sectors allocated to be used to replace sectors with a manufacturing defect.

High-Level / Logical Formatting

After partitioning, the second step is logical formatting (or creation of a file system). In this step, the operating system stores the initial file-system data structures onto the disk. These data structures may include maps of free and allocated space (a FAT or inodes) and an initial empty directory.

5.14 Disk Performance

Execution of a disk operation involves:

1. **Wait Time:** Time the process waits to be granted device access.
   - a. **Wait for device:** Time the request spent in wait queue.
   - b. **Wait for channel:** Time until a shared I/O channel is available.
2. **Access time:** Time the hardware needs to position the head.
   - a. **Seek time:** Time to position the head at the desired track.
   - b. **Rotational delay / latency:** Time for the desired sector to come under the head.
3. **Transfer time:** Time needed to transfer sectors to be read / written.

Estimating Access Time

Let, Average Seek Time = T_s

Rotational Speed = r rpm

$\therefore$ Rotational Delay = $\frac{1}{r}$ minute

$\therefore$ Average Rotational Delay = $\frac{1}{2r}$ minute

Let, there are N bytes per track, and we want to transfer b bytes.

$\therefore$ Transfer time of N bytes = $\frac{1}{r}$ minute

$\therefore$ Transfer time of b bytes = $\frac{b}{rN}$ minute

$\therefore$ Average Access Time for b bytes, T_a = Avg Seek Time + Avg Rotational Delay + Transfer Time

$$= T_s + \frac{1}{2r} + \frac{b}{rN}$$

Comparison of Disk Performance on Reading Fragmented and Contiguous Files

Let, T_s = 2 ms, r = 10,000 rpm.

Let, there are 512 bytes per sector and 320 sectors per track.

We need to read a file with 2560 sectors (= 1.3 MB).

Access Time When the File is Stored Compactly (Contiguous File)

To read the file, 8 contiguous tracks (2560 / 320 = 8) need to be read.

∴ Average access time to read *first* track

= avg. seek time + avg. rotational delay + time to read 320 sectors

$= T_s + \frac{1}{2r} + \frac{1}{r}$

= 2 ms + 3 ms + 6 ms

= 11 ms

∴ Average access time to read the *remaining* tracks = (2 + 6) × 7 ms = 56 ms

∴ Average access time to read all the sectors = 11 + 56 = 67 ms

Access Time When the File Sectors are Distributed Randomly Over the Disk (Fragmented File)

Average access time to read *any* sector

= avg. seek time + avg. rotational delay + time to read 1 sector

$= T_s + \frac{1}{2r} + \frac{1}{r \times 320}$

= 2 ms + 3 ms + 0.01875 ms

= 5.01875 ms

∴ Average access time to read all the sectors = 5.01875 × 2560 ms = 12848 ms

5.15 Disk Arm Scheduling Algorithms

For a multiprogramming system with many processes, the disk queue may often have several pending requests. Thus, when one request is completed, the operating system chooses which pending request to service next. How does the operating system make this choice? Any one of several disk-scheduling algorithms can be used, and we discuss them next.

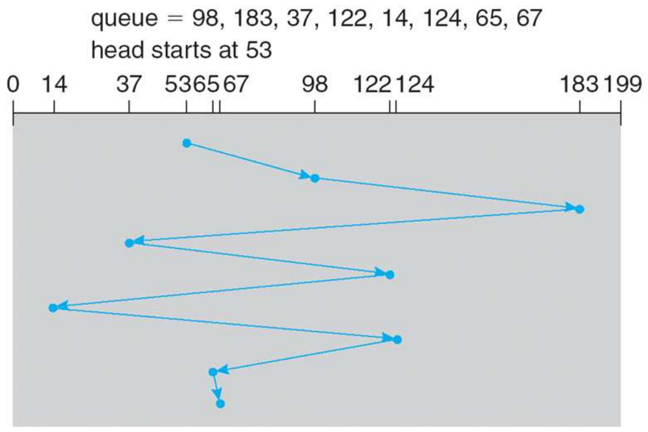
For most disks, the seek time dominates the other two times, so our target is to reduce the mean seek time to improve system performance substantially.

First-Come First-Serve (FCFS)

Requests are served in first-come first-serve manner.

Example

The disk queue contains the cylinder numbers containing the desired blocks.

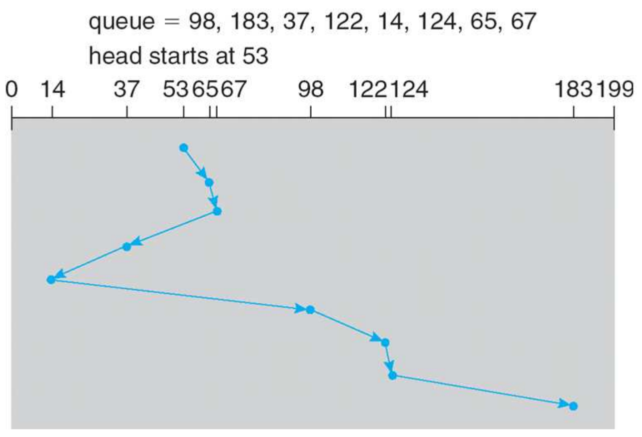


Total head movement = 640 cylinders

Shortest Seek First (SSF) / Shortest Seek Time First (SSTF)

Selects the request with the minimum seek time from the current head position.

Example



Total head movement = 236 cylinders

Disadvantages:

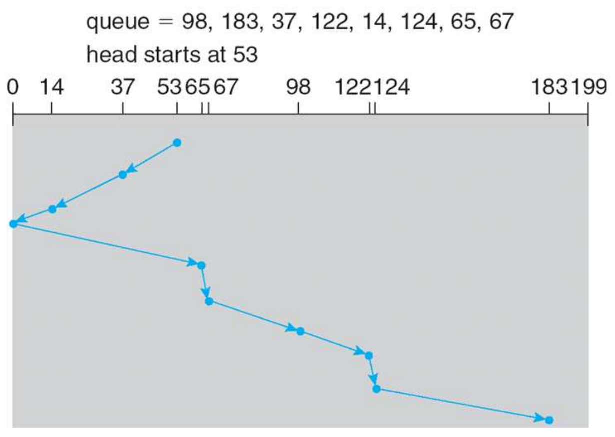
1. Although the SSTF algorithm is a substantial improvement over the FCFS algorithm, it is not optimal. In the example, we can do better by moving the head from 53 to 37, even though the latter is not closest, and then to 14, before turning around to service 65, 67, 98, 122, 124, and 183. This strategy reduces the total head movement to 208 cylinders.
2. With a heavily loaded disk, the arm will tend to stay in the middle of the disk most of the time, so requests at either extreme will have to wait until a statistical fluctuation in the load causes there to be no requests near the middle. Requests far from the middle may get poor service. The goals of minimal response time and fairness are in conflict here.

SCAN Scheduling / Elevator Algorithm

The disk arm starts at one end of the disk and moves toward the other end, servicing requests as it reaches each cylinder, until it gets to the other end of the disk. At the other end, the direction of head movement is reversed, and servicing continues. The head continuously scans back and forth across the disk.

The SCAN algorithm is sometimes called the *elevator algorithm*, since the disk arm behaves just like an elevator in a building, first servicing all the requests going up and then reversing to service requests the other way.

Example:



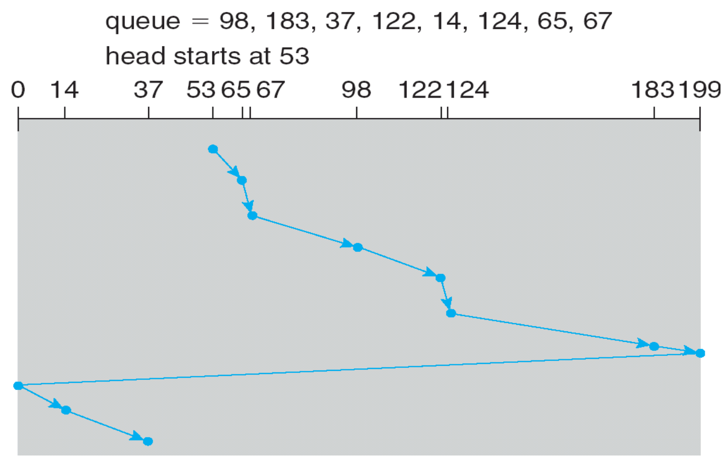
Disadvantage:

Assuming a uniform distribution of requests for cylinders, consider the density of requests when the head reaches one end and reverses direction. At this point, relatively few requests are immediately in front of the head, since these cylinders have recently been serviced. The heaviest density of requests is at the other end of the disk. These requests have also waited the longest, so why not go there first?

Circular SCAN (C-SCAN) / Modified Elevator

Like SCAN, C-SCAN moves the head from one end of the disk to the other, servicing requests along the way. When the head reaches the other end, however, it immediately returns to the beginning of the disk, without servicing any requests on the return trip.

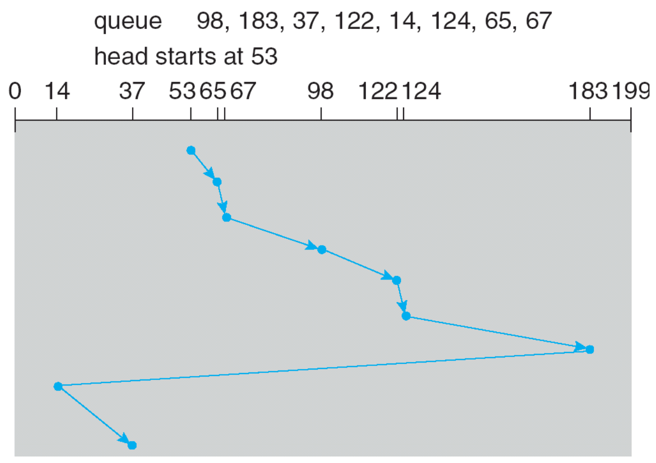
Example:



LOOK and C-LOOK

Both SCAN and C-SCAK move the disk arm across the full width of the disk. In practice, neither algorithm is often implemented this way. More commonly, the arm goes only as far as the final request in each direction. Then, it reverses direction immediately, without going all the way to the end of the disk. Versions of SCAN and C-SCAN that follow this pattern are called LOOK and C-LOOK scheduling, because they look for a request before continuing to move in a given direction.

Example:



Selection of a Disk-Scheduling Algorithm

Either SSTF or LOOK is a reasonable choice for the default algorithm.

5.16 Error Handling

Cause of Disk Errors

Disk manufacturers are constantly pushing the limits of the technology by increasing linear bit densities. Unfortunately, it is not possible to manufacture a disk to such specifications without defects. Manufacturing defects introduce bad sectors, that is, sectors that do not correctly read back the value just written to them. If the defect is very small, say, only a few bits, it is possible to use the bad sector and just let the ECC correct the errors every time. If the defect is bigger, the error cannot be masked.

Handling Disk Errors

All hard disks have some number of spare sectors allocated to be used to replace sectors with a manufacturing defect. For each bad sector, one of the spares is substituted for it.

There are two ways to do this substitution.

1. Remapping one of the spares as the bad sector. [As in *figure* 5.16(*b*)]
2. Shifting all the sectors up one. [As in *figure* 5.16(*c*)]

In both cases, the controller has to know which sector is which. It can keep track of this information through internal tables (one per track) or by rewriting the preambles to give the remapped sector numbers. If the preambles are rewritten, the method of *figure* 5.16(*c*) is more work (because 23 preambles must be rewritten) but ultimately gives better performance because an entire track can still be read in one rotation.

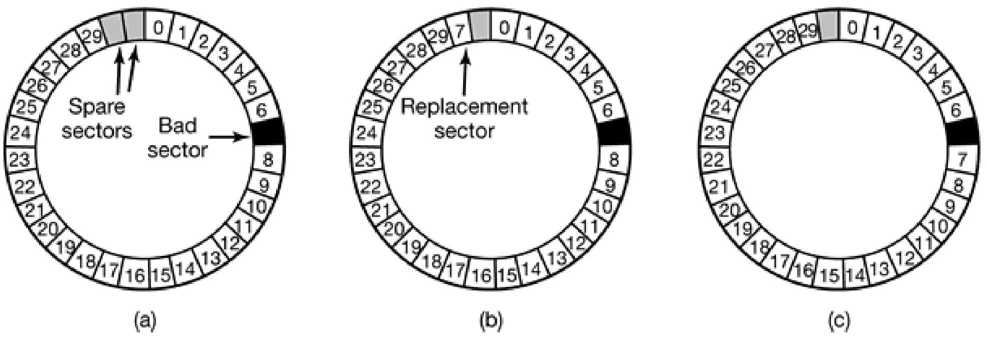


Figure 5.16: (a) A disk track with a bad sector.
(b) Substituting a spare for the bad sector.
(c) Shifting all the sectors to bypass the bad one.

5.17 Stable Storage

For some applications, it is essential that data never be lost or corrupted, even in the face of disk and CPU errors. Ideally, a disk should simply work all the time with no errors. Unfortunately, that is not achievable. What is achievable is a disk subsystem that has the following property: when a write is issued to it, the disk either correctly writes the data or it does nothing, leaving the existing data intact. Such as system is called *stable storage* and is implemented in software.

Stable storage uses a pair of identical disks with the corresponding blocks working together to form one error-free block. In the absence of errors, the corresponding blocks on both drives are the same. Either one can be read to get the same result.

What about in the presence of CPU crashes during stable writes? It depends on precisely when the crash occurs. There are five possibilities, as depicted in *figure* 5.17.

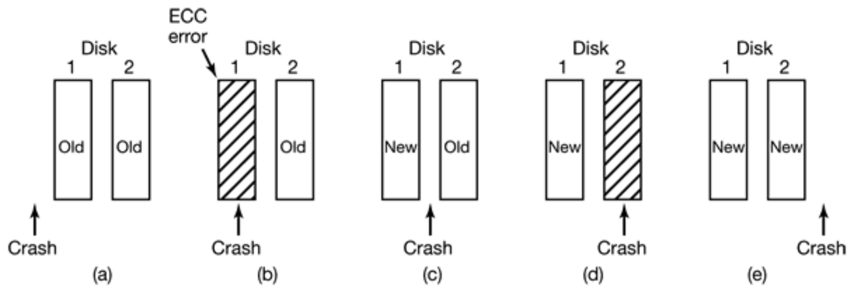


Figure 5.17: Analysis of the influence of crashes on stable writes.

In *figure* 5.17(*a*), the CPU crash happens before either copy of the block is written. During recovery, neither will be changed and the old value will continue to exist, which is allowed.

In *figure* 5.17(*b*), the CPU crashes during the write to drive l, destroying the contents of the block. However the recovery program detects this error and restores the block on drive 1 from drive 2. Thus the effect of the crash is wiped out and the old state is fully restored.

In *figure* 5.17(*c*), the CPU crash happens after drive 1 is written but before drive 2 is written. The point of no return has been passed here: the recovery program copies the block from drive 1 to drive 2. The write succeeds.

Figure 5.17(*d*) is like *figure* 5.17(*b*): during recovery, the good block overwrites the bad block. Again, the final value of both blocks is the new one.

Finally, in *figure* 5.17(*e*) the recovery program sees that both blocks are the same, so neither is changed and the write succeeds here too.

5.18 RAID (Redundant Array of Independent Disks)

Disk drives have continued to get smaller and cheaper, so it is now economically feasible to attach .many disks to a computer system. Having a large number of disks in a system presents opportunities for improving the rate at which data can be read or written, if the disks are operated in parallel. Furthermore, this setup offers the potential for improving the reliability of data storage, because redundant information can be stored on multiple disks. Thus, failure of one disk does not lead to loss of data. A variety of disk-organization techniques, collectively called redundant arrays of inexpensive disks (RAIDS), are commonly used to address the performance and reliability issues.

In the past, RAIDs composed of small, cheap disks were viewed as a cost-effective alternative to large, expensive disks; today, RAIDs are used for their higher reliability and higher data-transfer rate, rather than for economic reasons. Hence, the *I* in RAID now stands for "independent" instead of "inexpensive."

RAID Levels

RAID 0: Non-Redundant Striping

Striping at the level of blocks but without any redundancy (such as mirroring or parity bits).

Advantage: Improved read/write performance as the requests can be distributed among disks.

Disadvantage: Not reliable. If a single disk fails, all the information becomes corrupt.

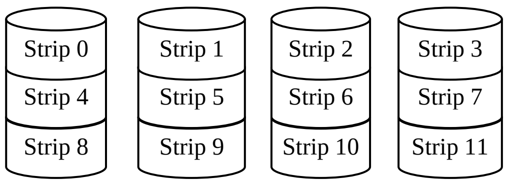


Figure: RAID 0.

RAID 1: Mirroring

Mirroring at the level of blocks.

Advantages:

1. Provides reliability.
2. Increased read performance.

Disadvantage: More expensive as the number of disks is doubled.

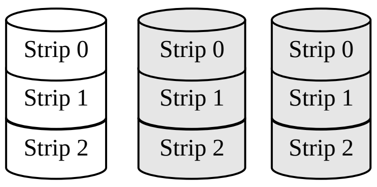


Figure: RAID 1 with 2 mirrored disks.

RAID 2: Memory-Style ECC

Error-correcting schemes store two or more extra bits and can reconstruct the data if a single bit is damaged. The idea of ECC can be used directly in disk arrays via striping of bytes across disks. For example, data is split into 4-bit nibbles and each bit is written in separate disks. Additionally, 3 hamming code (ECC) bits are written into 3 extra disks. If one of the disks fails, the remaining bits of the byte and the associated error-correction bits can be read from other disks and used to reconstruct the damaged data.

Advantage: Needs fewer disks than RAID 1.

Disadvantage: Synchronized spindles are needed to ensure each bit is available at the same time.

RAID2 is not used nowadays.

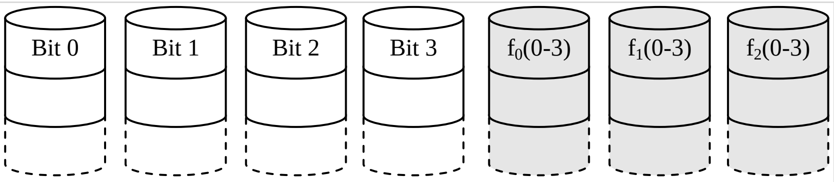


Figure: RAID 2.

RAID 3: Bit-Interleaved Parity

RAID level 3 improves on level 2 by taking into account the fact that unlike memory systems, disk controllers can detect whether a sector has been read correctly, so a single parity bit can be used for error correction as well as for detection.

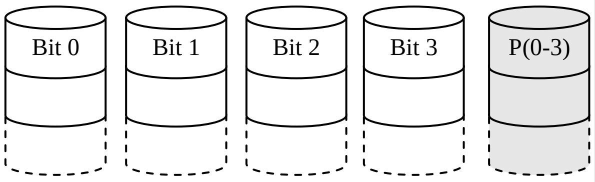


Figure: RAID 3.

Advantage: Needs fewer disks than RAID 1 or RAID 2.

Disadvantage: Synchronized spindles are needed to ensure each bit is available at the same time.

How Damaged Bit is Recovered

If one of the sectors is damaged, we know exactly which sector it is, and we can figure out whether any bit in the sector is a 1 or a 0 by computing the parity of the corresponding bits from sectors in the other disks (excluding the parity disk). If the parity of the remaining bits is equal to the stored parity, the missing bit is 0; otherwise, it is 1.

For example, consider the following:

Disk 1: 1
Disk 2: (*Lost*)
Disk 3: 1
Disk 4: 0
Parity: 0

$\therefore$ Parity of Disks 1, 3 and 4 = $1 \oplus 1 \oplus 0 = 0$ = *Stored Parity*
$\therefore$ The lost bit is 0.

RAID 4: Block-Interleaved Parity

The same as RAID 3 except that RAID 4 works at *block* level instead of *bit* level.

Advantages:

1. No synchronized spindles are needed.
2. Increased read performance.

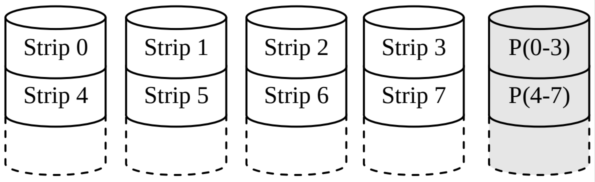


Figure: RAID 4.

Disadvantage: Small updates are a problem: requires two reads (old block + parity) and two writes (new block + parity) to update a disk block. Thus, parity disk may become a bottleneck.

RAID 5: Block-Interleaved Distributed Parity

Like RAID 4, except the parity is distributed on all disks.

Advantage: By spreading the parity across all the disks in the set, the potential update bottleneck of a single parity disk (occurring with RAID 4) is avoided.

Disadvantage: Reconstruction after failure is tricky.

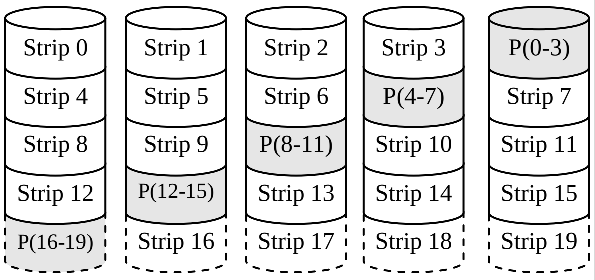


Figure: RAID 5.

RAID 6: P + Q Redundancy Scheme

RAID level 6 is much like RAID level 5 but stores extra redundant information to guard against multiple disk failures. Instead of parity, error-correcting codes such as the *Reed-Solomon codes* are used. In the scheme shown in Figure 12.11(g), 2 bits of redundant data are stored for every 4 bits of data – compared with 1 parity bit in level 5 – and the system can tolerate two disk failures.

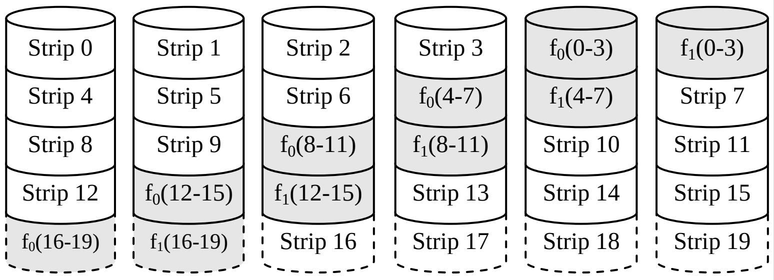


Figure: RAID 6.

RAID 0 + 1 / RAID 01 / RAID 1 + 0 / RAID 10

RAID level 0 + 1 (also known as RAID 1+ 0 / RAID 10 / RAID 01) refers to a combination of RAID levels 0 and 1. RAID 0 provides the performance, while RAID 1 provides the reliability.

Advantage: Better performance than RAID 5.

Disadvantage: More expensive as the number of disks is doubled.

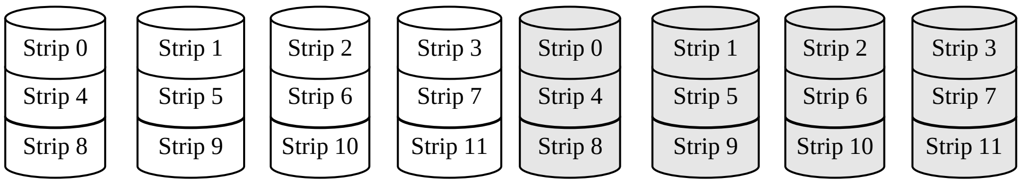


Figure: RAID 0 + 1.

HP AutoRAID

HP offers a disk array with a technology named AutoRAID. AutoRAID hardware and software monitor the use of data on a system to provide the best possible performance by determining the best RAID array level for that specific system. When the active workload is high, RAID 1 is used as it provides good read and write performance. For inactive data, RAID 5 is used as it provides low storage overheads.

CHAPTER 6
FILE MANAGEMENT

Roadmap and Concepts in Brief

- In this chapter, we'll learn how operating system manages files and directories.
- How can files be structured?
    1. Unstructured Stream (Pile)
    2. Sequential
    3. Indexed Sequential
    4. Direct or Hashed
- What are the different types of files?
    1. Regular Files
    2. Directories
    3. Block Special Files / Block Device Files
    4. Character Device Files
- How can directories be organized?
    1. Single-Level Directory Systems
    2. Two-Level Directory Systems
    3. Hierarchical / Tree Directory Systems
- How can files be implemented?
    1. Contiguous / Linear / Sequential Allocation
    2. Linked-List Allocation
    3. Linked-List Allocation Using a Table in Memory
    4. Inodes
- How can free blocks be kept track of?
    1. Free List Technique
    2. Bitmap / Bit Vector Technique
- How can long file names be handled?
    1. Setting a limit on file name length
    2. In-Line Method
    3. Heap Method

Theory

6.1 File Systems

The part of the operating system dealing with files is known as the *file system*.

Files are managed by the operating system. How they are structured, named, accessed, used, protected, and implemented are the issues of file management.

6.2 File Structure

1. Unstructured Stream (Pile)

- Unstructured sequence of bytes.
- Data are collected in the order they arrive.
- Records may have different fields.
- Purpose is to accumulate mass of data and save it.
- Record access is done by exhaustive search.

2. Sequential

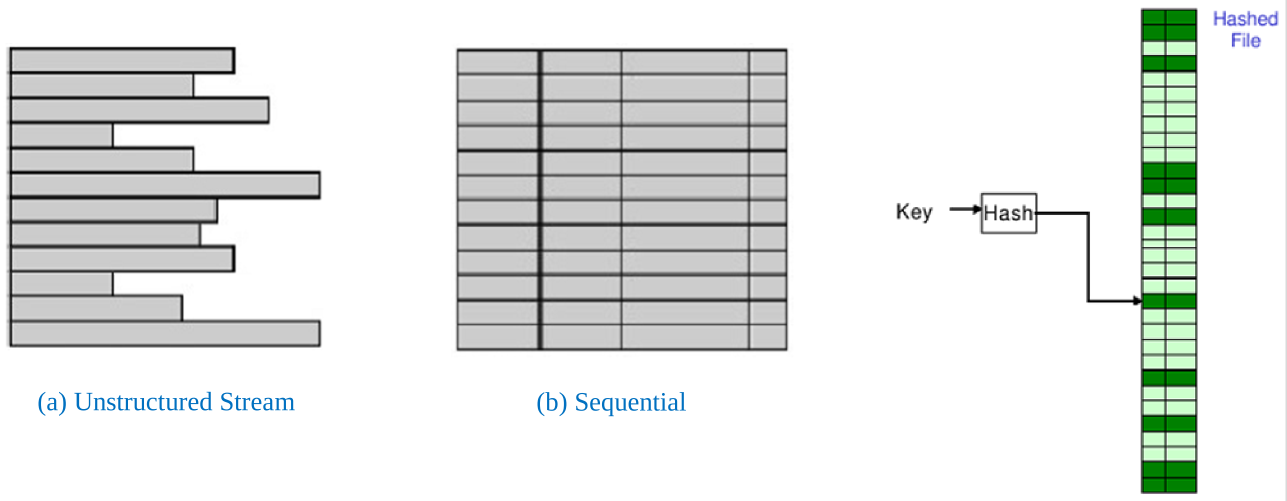
- Fixed format used for records.
- Records are the same length.

3. Indexed Sequential

- Inodes.

4. Direct or Hashed

- Key field required for each record.
- Key maps directly or via a hash mechanism to an address within a file.
- Directly access a block at a known address.



(a) Unstructured Stream

(b) Sequential

(c) Hashed

Figure 6.2: File Structures.

6.3 File Types

1. **Regular Files** – contains user information. Can be *ASCII* or *binary*.
2. **Directories** – system files for maintaining the structure of the file system.
3. **Block Special Files / Block Device Files** – used to model *disks*.
4. **Character Device Files** – related to input/output and used to model *serial I/O devices* such as terminals, printers and networks.

6.4 Directories

1. Single-Level Directory Systems

One directory contains all files.

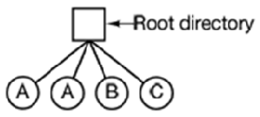


Figure 6.4(*a*):A single-level directory system containing four files, owned by three different people, *A*, *B* and *C*.

***Advantages*:**

a. Simplicity.
b. Ability to locate files quickly – there is only one place to look, after all!

***Disadvantage*:**

In a system with multiple users, if two users use the same names for their files, the latter user's file will overwrite the previous user's file.

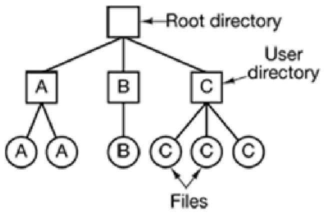


Figure 6.4(*b*): A two-level directory system. The letters indicate the owners of the directories and files.

2. Two-Level Directory Systems

Each user is given a private directory.

***Advantage*:**

No problem of using the same file name among different users.

***Disadvantage*:**

In order for the system to know which directory to search for a file, some kind of user login procedure is needed.

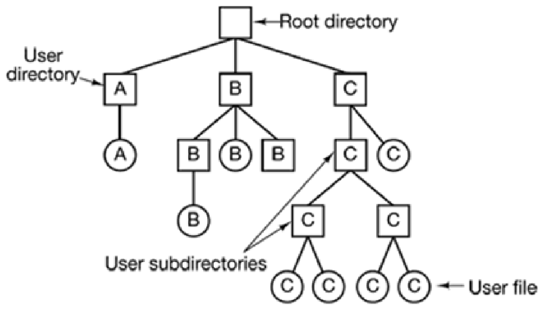


Figure 6.4(*c*): A hierarchical directory

3. Hierarchical / Tree Directory Systems

Tree of directories.

6.5 Implementing Files

1. Contiguous / Linear / Sequential 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.

We see an example of contiguous storage allocation in *figure* 6.6.1*a*. Here the first 40 disk blocks are shown, starting with block 0 on the left. Initially, the disk was empty. Then a file *A*, of length four blocks was written to disk starting at the beginning (block 0). After that a six-block file, *B*, was written starting right after the end of file *A*. Note that each file begins at the start of a new block, so that if file *A* was really 3½ blocks, some space is wasted at the end of the last block. In the figure, a total of seven files are shown, each one starting at the block following the end of the previous one. Shading is used just to make it easier to tell the files apart.

***Advantages*:**

1. It is simple to implement because keeping track of where a file's blocks are is reduced to remembering two numbers: the disk address of the first block and the number of blocks in the file.

   Given the number of the first block, the number of any other block can be found by a simple addition.

2. The read performance is excellent because the entire file can be read from the disk in a single operation. Only one seek is needed (to the first block). After that, no more seeks or rotational delays are needed so data come in at the full bandwidth of the disk. Thus contiguous allocation is simple to implement and has high performance.

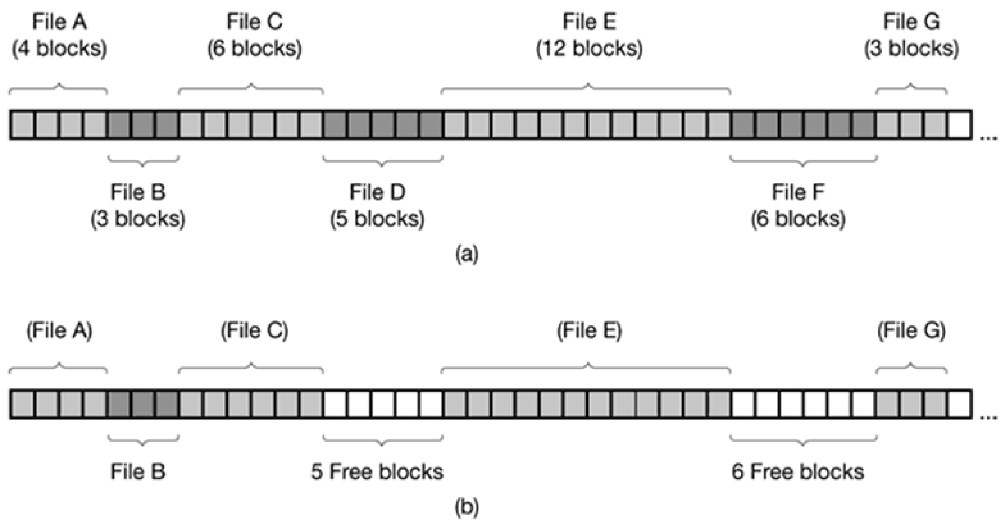


Figure 6.5.1: (a) Contiguous allocation of disk space for seven files.
(b) The state of the disk after files *D* and *F* have been removed.

***Disadvantage* – fragmentation:**

In time, the disk becomes fragmented. To see how this comes about, examine *figure* 6.6.1*b*. Here two files, *D* and *F* have been removed. When a file is removed, its blocks are freed, leaving a run of free blocks on the disk. The disk is not compacted on the spot to squeeze out the hole since that would involve copying all the blocks following the hole, potentially millions of blocks, As a result, the disk ultimately consists of files and holes, as illustrated in the figure.

***Use of contiguous space allocation*:**

There is one situation in which contiguous allocation is feasible and, in fact, widely 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.

2. Linked-List Allocation

The second method for storing files is to keep each one as a linked list of disk blocks, as shown in *figure* 6.6.2. The first *word* of each block is used as a pointer to the next one. The rest of the block is for data.

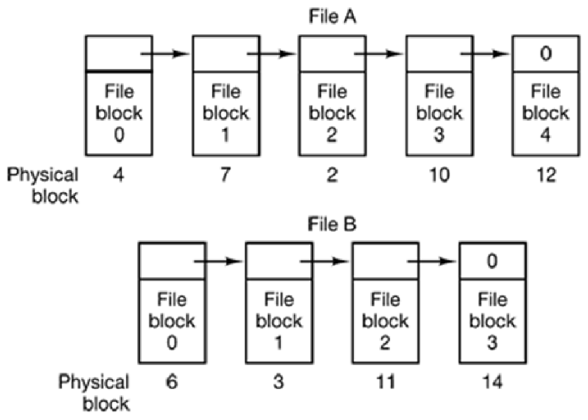


Figure 6.5.2: Storing a file as a linked list of disk blocks.

***Advantages*:**

1. Every disk block can be used in this method. No space is lost to disk fragmentation (except for internal fragmentation in the last block).
2. It is sufficient for the directory entry to merely store the disk address of the first block. The rest can be found starting there.

***Disadvantages*:**

1. 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.
2. The amount of data storage in a block is no longer a power of two because the pointer takes up a few bytes. While not fatal, having a peculiar size is less efficient because many programs read and write in blocks whose size is a power of two. With the first few bytes of each block occupied to a pointer to the next block, reads of the full block size require acquiring and concatenating information from two disk blocks, which generates extra overhead due to the copying.

3. 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*.

Figure 6.6.3 shows what the table looks like for the example of *figure* 6.6.2. In both figures, we have two files. File *A* uses disk blocks 4, 7, 2, 10, and 12, in that order, and file *B* uses disk blocks 6, 3, 11, and 14, in that order. Using the table of *figure* 6.6.3, we can start with block 4 and follow the chain all the way to the end. The same can be done starting with block 6. Both chains are terminated with a special marker (e.g., –1) that is not a valid block number.

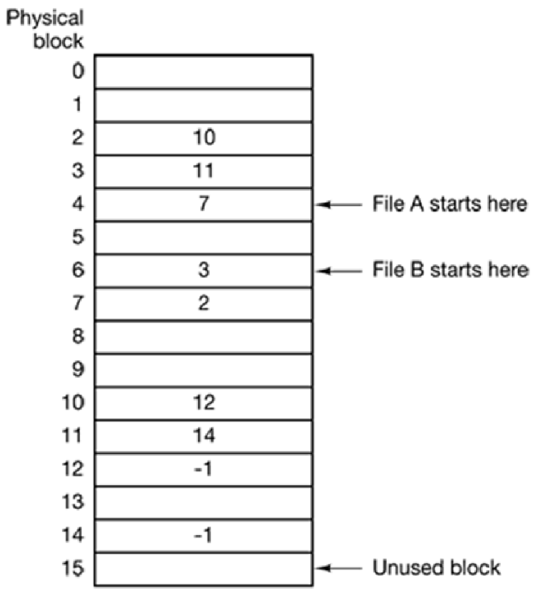


Figure 6.5.3: Linked list allocation using a file allocation table in main memory.

Such a table in main memory is called a **FAT** (*File Allocation Table*).

***Advantages*:**

1. The entire block is available for data.
2. Random access is much easier. Although the chain must still be followed to find a given offset within the file, the chain is entirely in memory, so it can be followed without making any disk references.
3. Like the previous method, it is sufficient for the directory entry to keep a single integer (the starting block number) and still be able to locate all the blocks, no matter how large the file is.

***Disadvantage*:**

The primary disadvantage of this method is that the entire table must be in memory all the time to make it work. With a 20-GB disk and a 1-KB block size, the table needs 20 million entries, one for each of the 20 million disk blocks. Each entry has to be a minimum of 3 bytes. For speed in lookup, they should be 4 bytes. Thus the table will take up 60 MB or 80 MB of main memory all the time, depending on whether the system is optimized for space or time.

Conceivably the table could be put in pageable memory, but it would still occupy a great deal of virtual memory and disk space as well as generating extra paging traffic.

4. Inodes (Index-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 *inode* (*index-node*), which lists the attributes and disk addresses of the files blocks.

The inodes contain some file attributes. The attributes contain the file size, three times (creation, last access, and last modification), owner, group, protection information, and a count of the number of directory entries that point to the inode.

Keeping track of disk blocks is done as shown in *figure* 6.6.4. The first 12 disk addresses are stored in the inode itself, so for small files, all the necessary information is right in the inode, which is fetched from disk to main memory when the file is opened. For somewhat larger files, one of the addresses in the inode is the address of a disk block called a *single indirect block*. This block contains additional disk addresses. If this still is not enough, another address in the inode, called a *double indirect block*, contains the address of a block that contains a list of single indirect blocks. Each of these single indirect blocks points to a few hundred data blocks. If even this is not enough, a *triple indirect block* can also be used.

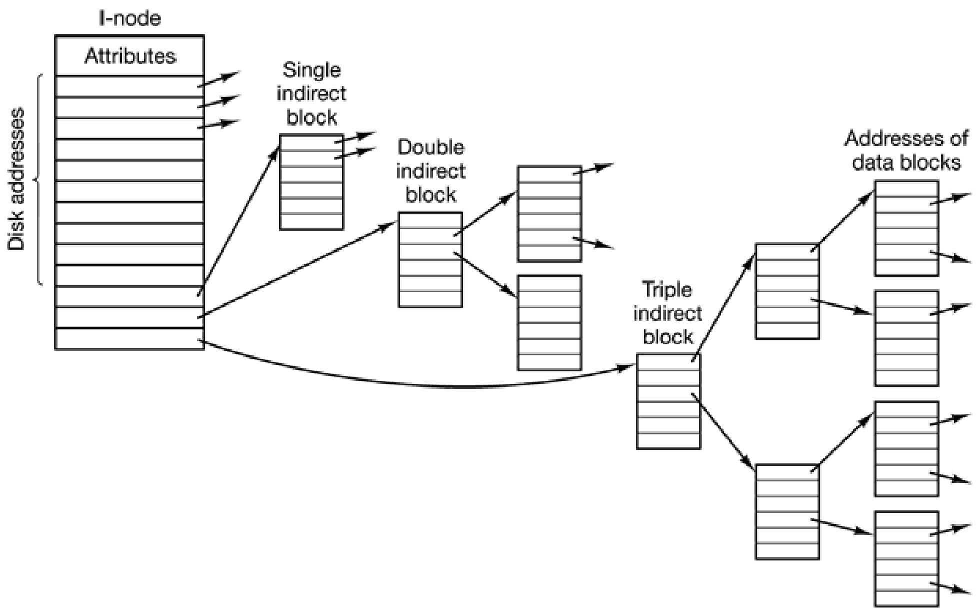


Figure 6.5.4: A UNIX i-node.

***Advantage*:**

The big advantage of this scheme over linked files using an in-memory table is that the inode need only be in memory when the corresponding file is open. If each inode occupies *n* bytes and a maximum of *k* files may be open at once, the total memory occupied by the array holding the inodes for the open files is only *kn* bytes. Only this much space need be reserved in advance.

Maximum file size for inode scheme

Let, the block numbers are of 4 bytes length, and block size is 1 KB.

Then, the number of addressable blocks will be

- Direct Blocks = 12
- Single Indirect Blocks = (1024 / 4) = 256
- Double Indirect Blocks = (256 × 256) = 65536
- Triple Indirect Blocks = (256 × 256 × 256) = 16777216

Therefore, the maximum file size will be:

(12 + 256 + 65536 + 16777216) × 1 KB = 16843020 KB = 16 GB

6.6 Implementing Directories in UNIX (using inode scheme)

A directory entry contains only two fields: the file name (14 bytes) and the number of the inode for that file (2 bytes), as shown in *figure* 6.7. These parameters limit the number of files per file system to 64K.

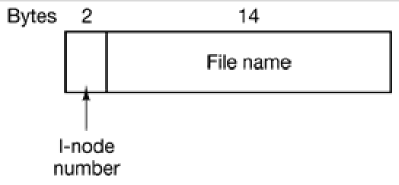


Figure 6.6: A UNIX directory entry.

6.7 File System Implementation – From User Space to Kernel Space

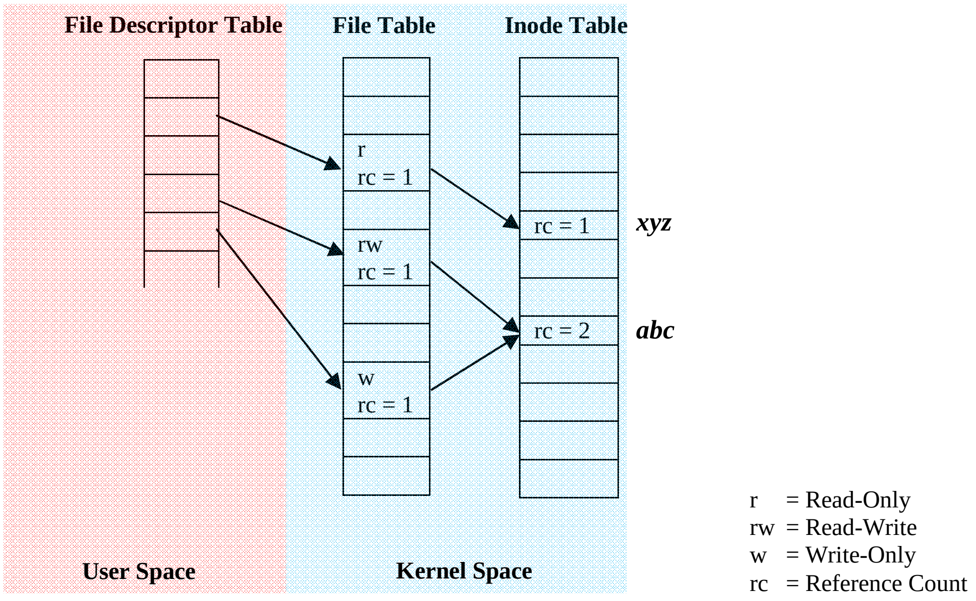
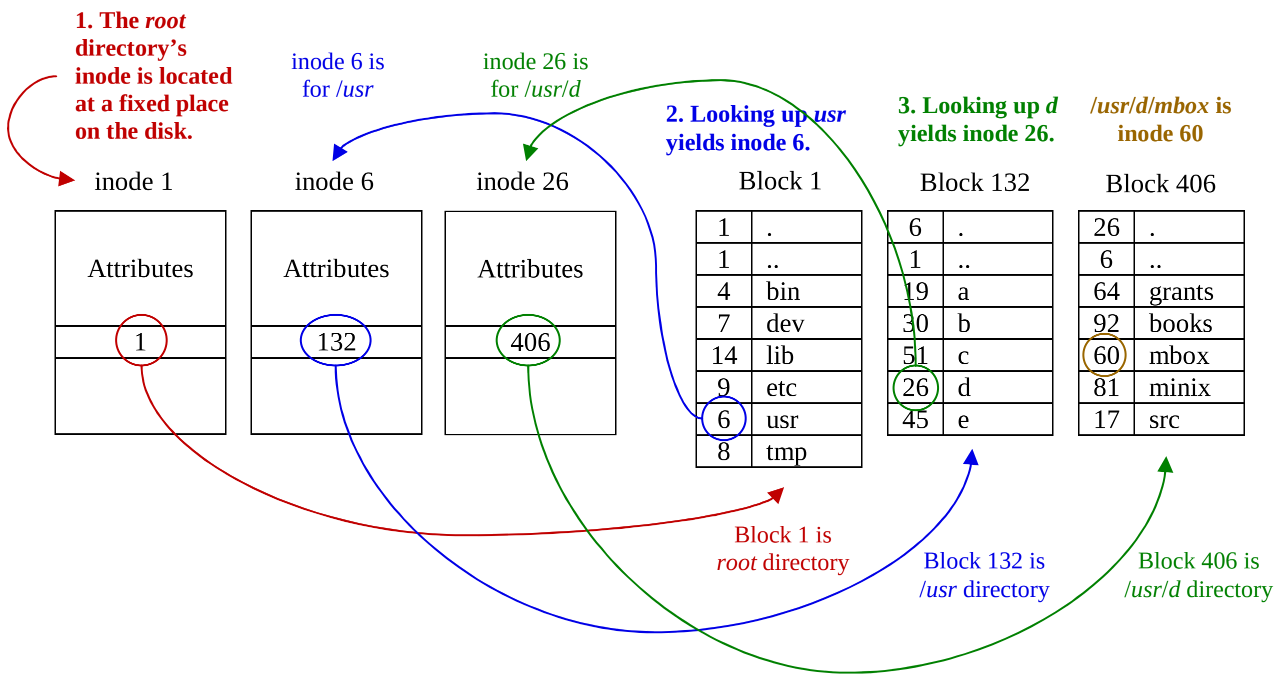


Figure 6.7: A process's file descriptor table, the kernel file table, and the inode table after the process has opened three files: *xyz* for read-only, *abc* for read-write, and *abc* again for read-only.

6.8 Looking up a File in UNIX

Let us consider how the path name */usr/d/mbox* is looked up. First, the file system locates the root directory. In UNIX its inode is located at a fixed place on the disk. From this inode, it locates the root directory which can be anywhere on the disk, but say block 1 in this case.

Then it reads the root directory and looks up the first component of the path, *usr*, in the root directory to find the inode number of the file */usr*. Locating an inode from its number is straightforward, since each one has a fixed location on the disk. From this inode, the system locates the directory for */usr* and looks up the next component, *d*, in it. When it has found the entry for *d*, it has the inode for the directory */usr/d*. From this inode it can find the directory itself and look up *mbox*. The inode for this file is then read into memory and kept there until the file is closed. The lookup process is illustrated in *figure* 6.9.


1. The *root* directory's inode is located at a fixed place on the disk.
inode 6 is for */usr*
inode 26 is for */usr/d*
2. Looking up *usr* yields inode 6.
3. Looking up *d* yields inode 26.
/usr/d/mbox is inode 60
inode 1
inode 6
inode 26
Block 1
Block 132
Block 406
Attributes
Attributes
Attributes
1
132
406
Block 1 is *root* directory
Block 132 is */usr* directory
Block 406 is */usr/d* directory

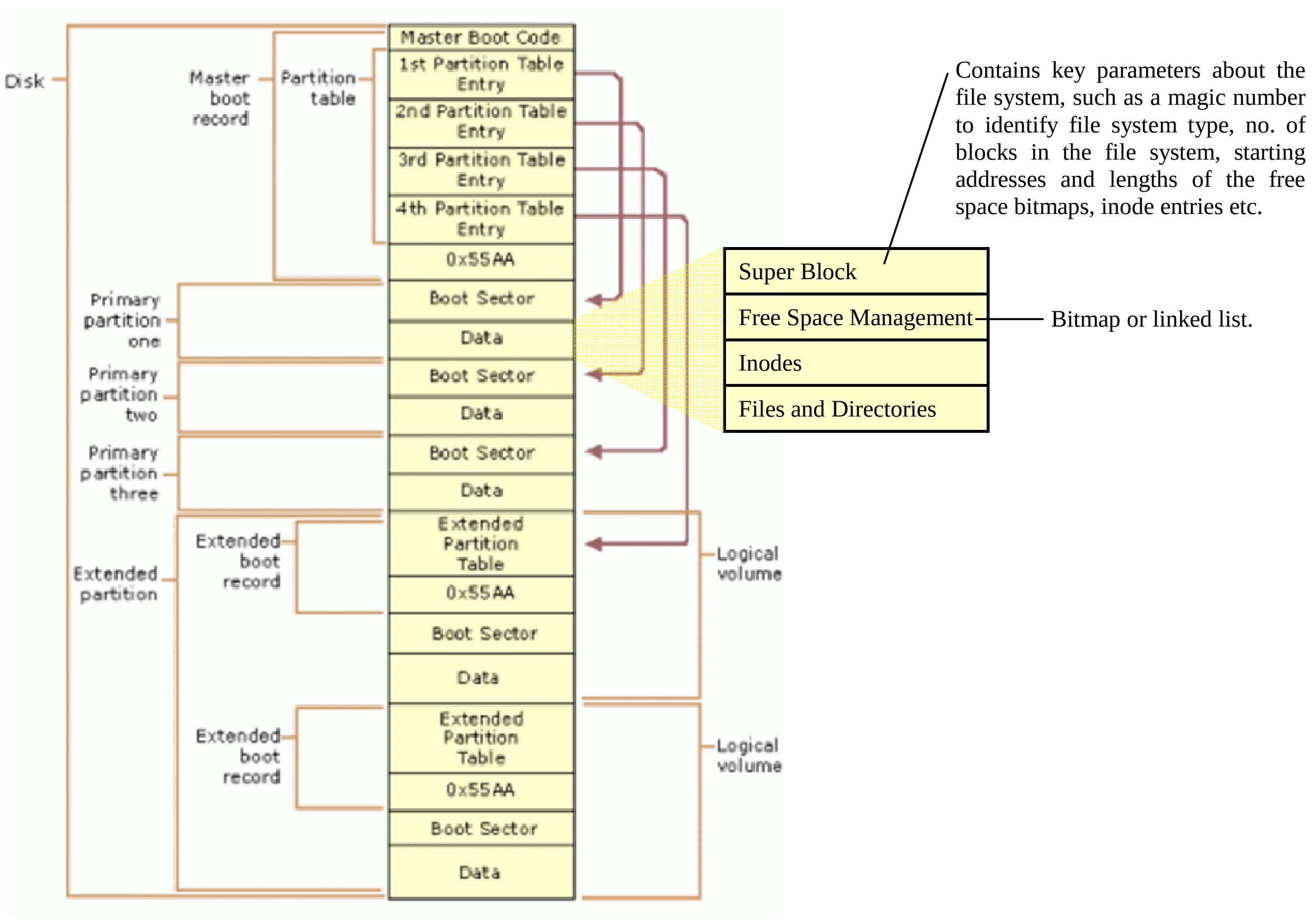
Block 1	
1	.
1	..
4	bin
7	dev
14	lib
9	etc
6	usr
8	tmp

Block 132	
6	.
1	..
19	a
30	b
51	c
26	d
45	e

Block 406	
26	.
6	..
64	grants
92	books
60	mbox
81	minix
17	src

Figure 6.8: Looking up a file in UNIX.

6.9 Typical File System Layout



6.10 Handling Long File Names

1. Set a limit on file name length, typically 255 characters, and then use the design of *figure* 6.7 with 255 characters reserved for each file name.

Advantage: Simple to implement.

Disadvantage: Wastes a great deal of directory space, since few files have such long names.

2. In-Line Method [*figure* 6.10 (*a*)]

Each directory entry contains a fixed header portion containing the length of the entry and other information. This fixed-length header is followed by the actual file name, however long it may be. Each file name is terminated by a special character (usually the *null character*), which is represented in *figure* 6.10 by a box with a cross in it. To allow each directory entry to begin on a word boundary, each file name is filled out to an integral number of words, shown by shaded boxes in the figure.

Disadvantages:

1. 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 the same one we saw with contiguous disk files, only now compacting the directory is feasible because it is entirely in memory.

2. A single directory entry may span multiple pages, so a page fault may occur while reading a file name.

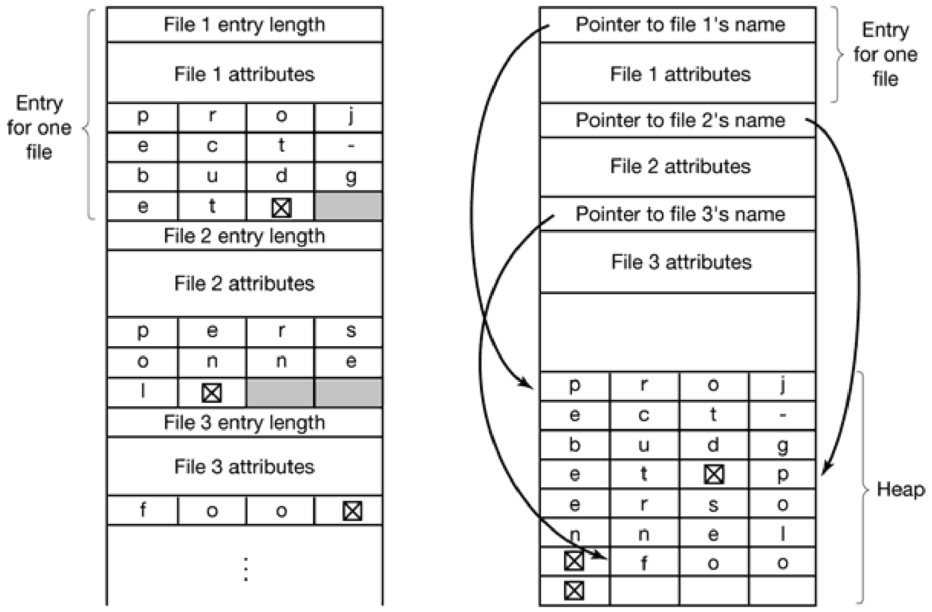
3. Heap Method [*Figure* 6.10 (*b*)]

Another way to handle variable-length names is to make the directory entries themselves all fixed length and keep the file names together in a heap at the end of the directory, as shown in *figure* 6.10 (*b*).

***Advantage*:**

When an entry is removed, the next file entered will always fit there.

Of course, the heap must be managed and page faults can still occur while processing file names. One minor win here is that there is no longer any real need for file names to begin at word boundaries, so no filler characters are needed after file names in *figure* 6.10 (*b*) and they are in *figure* 6.10 (*a*).



(a) In-line method. **(b)** Heap method.

Figure 6.10: Two ways of handling long file names.

6.11 Block Size

Why we need blocks

Storing a file as a contiguous sequence of bytes has the obvious problem that if a file grows, it will probably have to be moved on the disk. The same problem holds for segments in memory, except that moving a segment in memory is a relatively fast operation compared to moving a file from one disk position to another. For this reason, nearly all file systems chop files up into fixed-size blocks that need not be adjacent.

Factors affecting the choice of block size

1. Block size too large – Disk space will be wasted.
2. Block size too small – Poor performance.

Time needed to read a block

Reading each block normally requires a *seek* and a *rotational delay*.

As an example, consider a disk with 131,072 bytes per track, a rotation time of 8.33 msec, and an average seek time of 10 msec. The time in milliseconds to read a block of *k* bytes is then the sum of the seek, rotational delay, and transfer times:

$$10 + 4.165 + (k / 131072) \times 8.33$$

6.12 Keeping Track of Free Blocks

1. Free List Technique

A linked list of disk blocks is used, with each block holding as many free disk block numbers as will fit.

With a 1-KB block and a 32-bit disk block number, each block on the free list holds the numbers of 255 free blocks. (One slot is needed for the pointer to the next block).

A 16-GB disk needs a free list of maximum 16,794 blocks to hold all 2^{24} disk block numbers. Often free blocks are used to hold the free list.

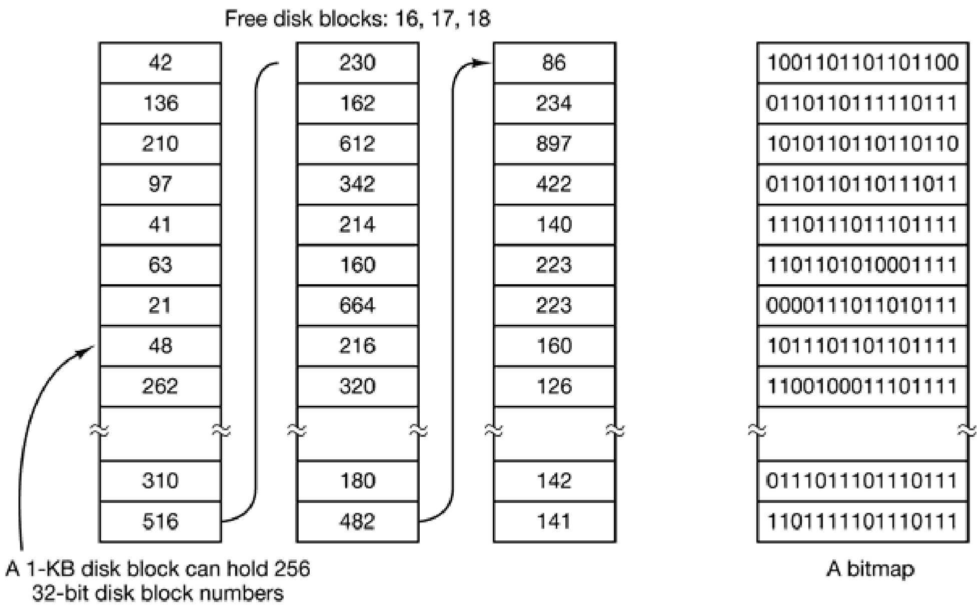
2. Bitmap / Bit Vector Technique

A disk with *n* blocks requires a bitmap with *n* bits. Free blocks are represented by 1s in the map, allocated blocks by 0s (or vice versa).

A 16-GB disk has 2^{24} 1-KB blocks and thus requires 2^{24} bits for the map, which requires 2048 blocks. It is not surprising that the bitmap requires less space, since it uses 1 bit per block, versus 32 bits in the linked list model.

For example, consider a disk where blocks 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17, 18, 25, 26 and 27 are free and the rest of the blocks are allocated. The free-space bitmap would be:

001111001111110001100000011100000…



(a) Free List **(b)** Bitmap

Figure 6.11: Keeping track of free blocks.

6.13 Virtual File System (VFS)

A virtual file system is a file system which provides a general interface to user programs to access different file-system-specific functions in the same machine.

A virtual file system implementation consists of three major layers as depicted schematically in *figure* 6.13.

The first layer is the file system interface, based on the `open()`, `read()`, `write()` and `close()` calls and on file descriptors.

The second layer is called the virtual file system (VFS) layer. It serves two important functions:

1. It separates file-system-generic operations from their file-system-specific implementation by defining a clean VFS interface. Several implementations for the VFS interface implemented by respective file systems may coexist on the same machine.
2. The VFS provides a mechanism for uniquely representing a file throughout a network. The VFS is based on a file-representation structure, called a *vnode*, that contains a numerical ID for a network-wide unique file.

The VFS activates file-system-specific operations to handle local requests according to their file-system types. File descriptors are constructed from the relevant vnodes and are passed as arguments to these (file-system-specific) procedures. The layer implementing the file-system type is the third layer of the architecture.

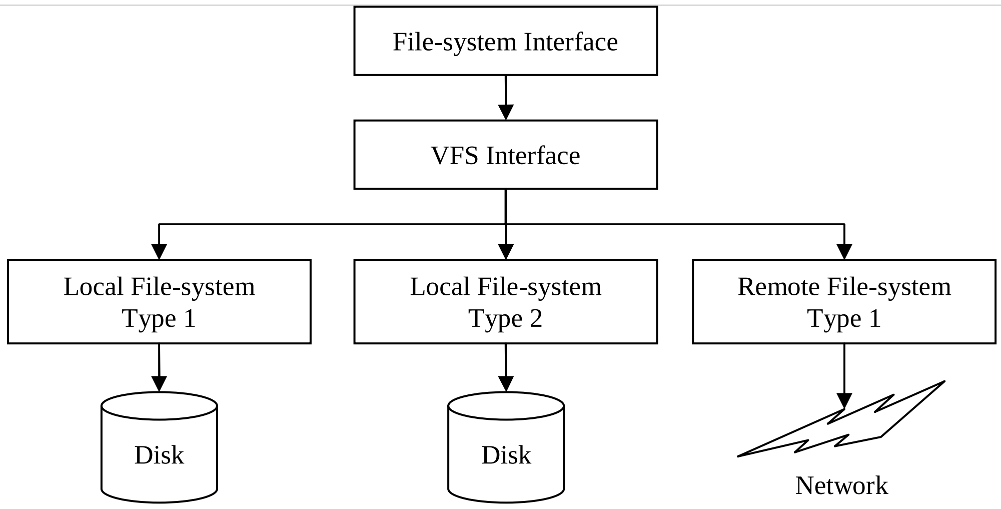


Figure 6.13: Schematic view of a VFS.

6.14 Shared Files / Symbolic Links / Hard Links

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. *Figure* 6.14 shows the file system of *figure* 6.4(*c*) again, only with one of *C*'s files now present in one of *B*'s directories as well. The connection between *B*'s directory and the shared file is called a *link* or *hard link*. The file system itself is now a *Directed Acyclic Graph* (DAG), rather than a tree.

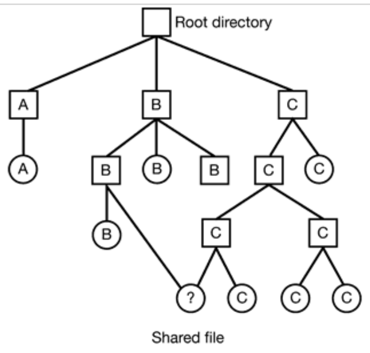


Figure 6.14: File system combining a shared file.

6.15 Access Control

The most common approach to the protection problem is to make access dependent on the identity of the user. Different users may need different types of access to a file or directory. The most general scheme to implement identity-dependent access is to associate with each file and directory an access-control list (ACL) specifying user names and the types of access allowed for each user. When a user requests access to a particular file, the operating system checks the access list associated with that file. If that user is listed for the requested access, the access is allowed. Otherwise, a protection violation occurs, and the user job is denied access to the file.

To condense the length of the access-control list, many systems recognize three classifications of users in connection with each file:

- **Owner.** The user who created the file is the owner.
- **Group.** A set of users who are sharing the file and need similar access is a group, or work group.
- **Universe / Others.** All other users in the system constitute the universe.

Below are some example file protection modes:

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

6.16 Disk Quotas

To prevent people from hogging too much disk space, multiuser operating systems often provide a mechanism for enforcing disk quotas. The idea is that the system administrator assigns each user a maximum allotment of files and blocks, and the operating system makes sure that the users do not exceed their quotas. A typical mechanism is described below.

When a user opens a file, the attributes and disk addresses are located and put into an open file table in main memory. Among the attributes is an entry telling who the owner is. Any increases in the file's size will be charged to the owner's quota.

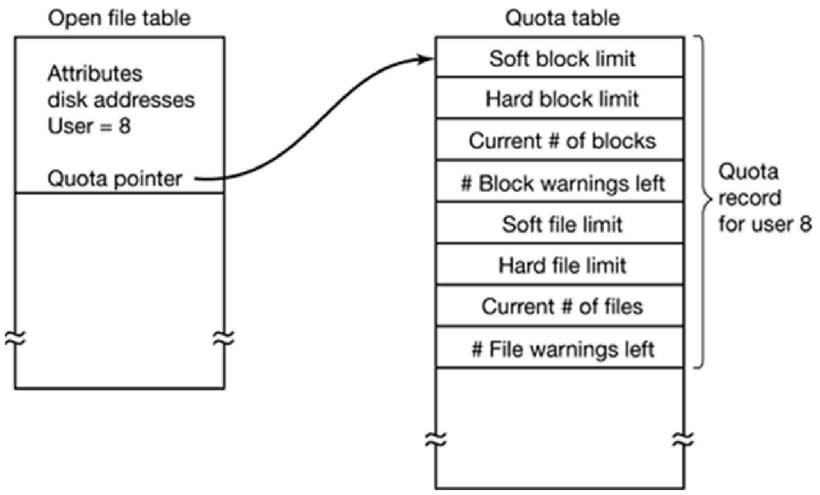


Figure 6.16: Quotas are kept track of on a per-user basis in a quota table.

A second table contains the quota record for every user with a currently open file, even if the file was opened by someone else. This table is shown in Fig. 6-23. It is an extract from a quota file on disk for the users whose files are currently open. When all the files are closed, the record is written back to the quota file.

When a user attempts to log in, the system checks the quota file to see if the user has exceeded the soft limit for either number of files or number of disk blocks. If either limit has been violated, a warning is displayed, and the count of warnings remaining is reduced by one. If the count ever gets to zero, the user has ignored the warning one time too many, and is not permitted to log in. Getting permission to log in again will require some discussion with the system administrator.

6.17 Log-Structured / Log-Based / Transaction-Oriented / Journaling File Systems

A system crash can cause inconsistencies among on-disk filesystem data structures, such as directory structures, free-block pointers, and free FCB (File Control Block, e.g., inode) pointers. A typical operation, such as file create, can involve many structural changes within the file system on the disk. Directory structures are modified, FCBs are allocated, data blocks are allocated, and the free counts for all of these blocks are decreased. These changes can be interrupted by a crash, and inconsistencies among the structures can result. For example, the free FCB count might indicate that an FCB had been allocated, but the directory structure might not point to the FCB. The FCB would be lost were it not for the consistency-check phase.

To repair these inconsistencies, log-based recovery techniques are applied to file-system metadata updates. File systems such as NTFS, JFS (Journaling File System) etc. use these techniques.

Fundamentally, all metadata changes are written sequentially to a log. Each set of operations for performing a specific task is a transaction. Once the changes are written to this log, they are considered to be committed, and the system call can return to the user process, allowing it to continue execution. Meanwhile, these log entries are replayed across the actual file-system structures. As the changes are made, a pointer is updated to indicate which actions have completed and which are still incomplete. When an entire committed transaction is completed, it is removed from the log file.

If the system crashes, the log file will contain zero or more transactions. Any transactions it contains were not completed to the file system, even though they were committed by the operating system, so they must now be completed. The transactions can be executed from the pointer until the work is complete so that the file-system structures remain consistent. The only problem occurs when a transaction was aborted – that is, was not committed before the system crashed. Any changes from such a transaction that were applied to the file system must be undone, again preserving the consistency of the file system. This recovery is all that is needed after a crash, eliminating any problems with consistency checking.

Chapter 9
Security

Theory

9.1 Attacks From Inside the System

Trojan Horses

A *Trojan horse* is a seemingly innocent program containing code to perform an unexpected and undesirable function. This function might be modifying, deleting or encrypting the user's files, copying them to a place where the cracker can retrieve them later, or even sending them to the cracker or a temporary safe hiding place via email or FTP. To have the Trojan horse run, the person planting it first has to get the program carrying it executed.

Login Spoofing

Login spoofing refers to writing a program that emulates a log-in screen. When a user sits down and types a login name, the program responds by asking for a password and disabling echoing. After the login name and password have been collected, they are written away to a file and the phony login program sends a signal to kill its shell. This action logs the malware out and triggers the real login program to start and display its prompt. The user assumes that he made a typing error and just logs in again. This time it works. But in the meantime, the malware has acquired another (*login na*me, *password*) pair. By logging in at many terminals and starting the login spoofer on all of them, it can collect many passwords.

Logic Bombs

Logic bomb is a piece of code written by one of a company's (currently employed) programmers and secretly inserted into the production operating system. As long as the programmer feeds it its daily password, it does nothing. However, if the programmer is suddenly fired and physically removed from the premises without warning, the next day (or next week) the logic bomb does not get fed its daily password, so it goes off. Many variants on this theme are also possible. In one famous case, the logic bomb checked the payroll. If the personnel number of the programmer did not appear in it for two consecutive payroll periods, it went off. Going off might involve clearing the disk, erasing files at random, carefully making hard-to-detect changes to key programs, or encrypting essential files.

Trap Doors

Trap doors are pieces of code that are inserted into the system by a system programmer to bypass some normal checks. For example, a programmer could add code to the login program to allow anyone to log in using the login name "*abcde*" no matter what was in the password file. If this trap door code were inserted by a programmer working for a computer manufacturer and then shipped with its computers, the programmer could log into any computer made by his company, no matter who owned it or what was in the password file. The trap door simply bypasses the whole authentication process.

Buffer Overflows

Virtually all operating systems and most systems programs are written in the C programming language. Unfortunately, no C compiler does array bounds checking. This property of C leads to attacks of the following kind. In *figure* 9.1(*a*), we see the main program running, with its local variables on the stack. At some point it calls a procedure *A*, as shown in *figure* 9.1(*b*). The standard calling sequence starts out by pushing the return address, (which points to the instruction following the call) onto the stack. It then transfers control to *A*, which decrements the stack pointer to allocate storage for its local variables.

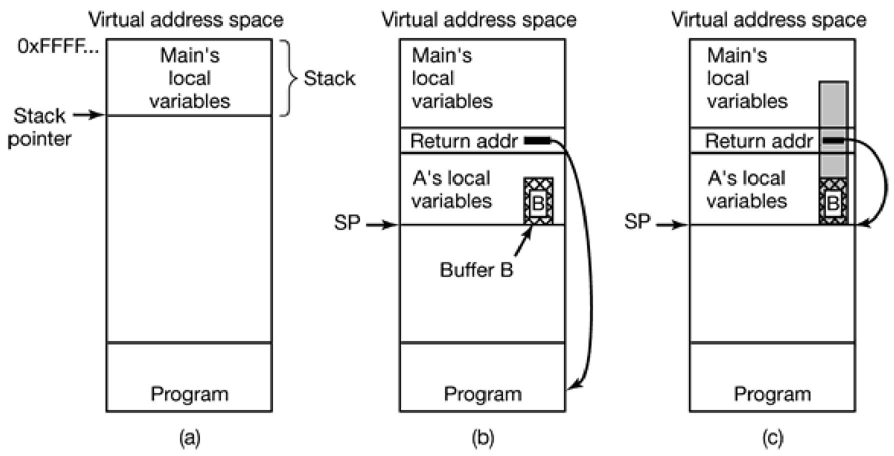


Figure 9.1: (a) Situation when the main program is running.
(b) After the procedure *A* has been called.
(c) Buffer overflow shown in gray.

Suppose that the job of *A* requires acquiring the full file path (possibly by concatenating the current directory path with a file name) and then opening it or doing something else with it. *A* has a fixed-size buffer (i.e., array) *B* to hold a file name, as shown in *figure* 9.1(*b*). Suppose that the user of the program provides a file name that is 2000 characters long. When the file name is used it will fail to open, but the attacker does not care. When the procedure copies the file name into the buffer, the name overflows the buffer and overwrites memory as shown in the gray area of *figure* 9.1(*c*). Worse yet, if the file name is long enough, it also overwrites the return address, so when *A* returns, the return address is taken from the middle of the file name. If this address is random junk, the program will jump to a random address and probably crash within a few instructions.

But what if the file name does not contain random junk? What if it contains a valid binary program and the layout has been very, very carefully made so that the word overlaying the return address just happens to be the address of the start of the program, for example, the address of *B*? What will happen is that when *A* returns, the program now in *B* will start executing. In effect, the attacker has inserted code into the program and gotten it executed.

Attacks From Outside the System

Virus (Vital Information Resources Under Siege)

A *virus* is a program that can reproduce itself by attaching its code to another program. In addition, the virus can also do other things in addition to reproducing itself.

Worms

Worms are like viruses but are self replicating rather than attaching their code.

9.2 How Viruses Work

Companion Viruses

A *companion virus* does not actually infect a program, but gets to run when the program is supposed to run. For example, the Windows desktop contains shortcuts (symbolic links) to programs. A virus can change the target of a shortcut to make it point to the virus. When the user double clicks on an icon, the virus is executed. When it is done, the virus just runs the original target program.

Executable Program Viruses

One step up in complexity are viruses that infect executable programs. The simplest of these viruses just overwrites the executable program with itself. These are called **overwriting viruses**.

The problem with an overwriting virus is that it is too easy to detect. Consequently, most viruses attach themselves to the program and do their dirty work, but allow the program to function normally afterward. Such viruses are called **parasitic viruses**.

Parasitic viruses can attach themselves to the front, the back, or the middle of the executable program, changing the starting address field in the header to point to the start of the virus. After executing the virus, the control is handed over to the actual program.

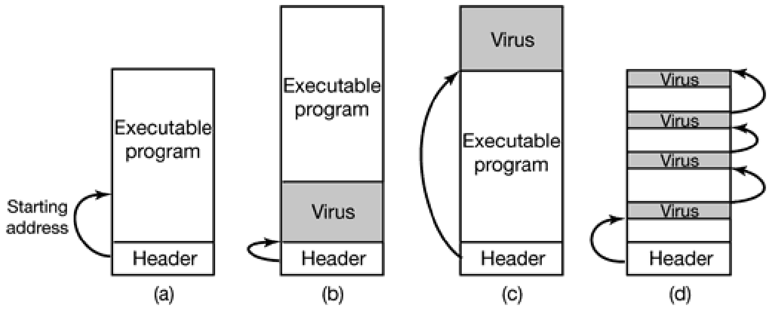


Figure 9.2: (a) An executable program.
(b) With a virus at the front.
(c) With a virus at the end.
(d) With a virus spread over free space within the program.

Memory Resident Viruses

So far we have assumed that when an infected program is executed, the virus runs, passes control to the real program, and exits. In contrast, a *memory-resident virus* stays in memory all the time, either hiding at the very top of memory or perhaps down in the grass among the interrupt vectors, the last few hundred bytes of which are generally unused.

A typical memory-resident virus captures one of the trap or interrupt vectors by copying the contents to a scratch variable and putting its own address there, thus directing that trap or interrupt to it. The best choice is the system call trap. In that way, the virus gets to run (in kernel mode) on every system call. When it is done, it just invokes the real system call by jumping to the saved trap address.

Boot Sector Viruses

A *boot sector virus*, [which includes MBR (Master Boot Record) viruses], first copies the true boot sector to a safe place on the disk so it can boot the operating system when it is finished.

Macro Viruses

Many programs, such as *Word* and *Excel*, allow users to write macros to group several commands that can later be executed with a single keystroke. Macros can also be attached to menu items, so that when one of them is selected, the macro is executed. In Microsoft *Office*, macros can contain entire programs in Visual Basic, which is a complete programming language. Since macros may be document specific, *Office* stores the macros for each document along with the document.

Now, if a document written in *Word* contains a macro virus attached to the OPEN FILE function, then opening the document will cause the virus to execute.

9.3 How Virus Scanners Work

Matching Virus Database Records

Every executable file on the disk is scanned looking for any of the viruses in the database of known viruses.

Using Fuzzy Search

Since minor variants of known viruses pop up all the time, a fuzzy search is needed, so a 3-byte change to a virus does not let it escape detection. However, fuzzy searches are not only slower than exact searches, but they may turn up false alarms, that is, warnings about legitimate files that happen to contain some code vaguely similar to a virus. The more viruses in the database and the broader the criteria for declaring a hit, the more false alarms there will be. If there are too many, the user will give up in disgust. But if the virus scanner insists on a very close match, it may miss some modified viruses. Getting it right is a delicate heuristic balance. Ideally, the lab should try to identify some core code in the virus that is not likely to change and use this as the virus signature to scan for.

Matching File Length

Another way for the antivirus program to detect file infection is to record and store on the disk the lengths of all files. If a file has grown since the last check, it might be infected.

Integrity Checking

A completely different approach to virus detection is *integrity checking*. An antivirus program that works this way first scans the hard disk for viruses. Once it is convinced that the disk is clean, it computes a checksum for each executable file and writes the list of checksums for all the relevant files in a directory to a file, *checksum*, in that directory. The next time it runs, it re-computes all the checksums and sees if they match what is in the file *checksum*. An infected file will show up immediately.

Behavioral Checking

Another strategy used by antivirus software is *behavioral checking*. With this approach, the antivirus program lives in memory while the computer is running and catches all system calls itself. The idea is that it can then monitor all activity and try to catch anything that looks suspicious. For example, no normal program should attempt to overwrite the boot sector, so an attempt to do so is almost certainly due to a virus. Likewise, changing the flash ROM is highly suspicious.