Operating System

Reference Books:

1. Operating System Concepts 7th Edition, By galvin

This PDF contains the notes from the standards books and are only meant for GATE CSE aspirants.

Notes Compiled By-
Manu Thakur
Mtech CSE, IIT Delhi
worstguymanu@gmail.com
https://www.facebook.com/Worstguymanu

Operating System

Multiprogramming:

Multiprogramming increases CPU utilization by organizing jobs (code and data) so that the CPU always has one to execute. The operating system keeps several jobs in memory simultaneously (Figure 1.7). This set of jobs can be a subset of the jobs kept in the job pool—which contains all jobs that enter the system—since the number of jobs that can be kept simultaneously in memory is usually smaller than the number of jobs that can be kept in the job pool. The operating system picks and begins to execute one of the jobs in memory.

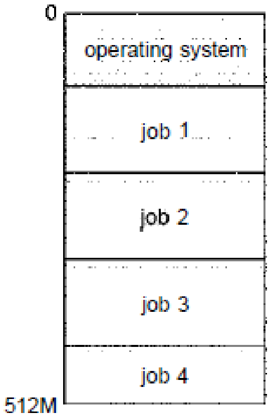


Figure 1.7 Memory layout for a multiprogramming system.

Eventually, the job may have to wait for some task, such as an I/O operation, to complete. In a **non-Multiprogrammed system(Batch Operating Systems)**, the **CPU would sit idle**. In a **Multiprogrammed system**, the operating system simply switches to, and executes, another job. When that job needs to wait, the CPU is switched to another job, and so on. Eventually, the first job finishes waiting and gets the CPU back. As long as at least one job needs to execute, the CPU is never idle.

Note: Multiprogrammed systems provide an environment in which the various system resources (for example, CPU, memory, and peripheral devices) are utilized effectively, but they do not provide for user interaction with the computer system.

Time sharing (or multitasking):

It is a **logical extension of multiprogramming**. In time-sharing systems, the CPU executes multiple jobs by switching among them, but the switches occur so frequently that the users can interact with each program while it is running.

Time sharing requires an interactive (or hands-on) computer system, which provides direct communication between the user and the system. The user gives instructions to the operating system or to a program directly, using an input device such as a keyboard or a mouse, and waits for immediate results on an output device. Accordingly, the response time should be short—typically less than one second. A time-shared operating system allows many users to share the computer simultaneously. A time-shared operating system uses CPU scheduling and multiprogramming to provide each user with a small portion of a time-shared computer. **Each user has at least one separate program in memory**.

Note: the **jobs are kept initially on the disk in the job pool**. This pool consists of all processes residing on disk awaiting allocation of main memory. If several jobs are ready to be brought into memory, and if there is not enough room for all of them, then the system must choose among them. Making this decision is **job scheduling**. When the operating system selects a job from the job pool, it loads that job into memory for execution. Having several programs in memory at the same time requires some form of memory management. If several jobs are ready to run at the same time, the system must choose among them. Making this decision is **CPU scheduling**.

Dual-Mode Operation:

In order to ensure the proper execution of the operating system, we must be able to distinguish between the execution of operating-system code and user-defined code. We need two separate modes of operation: user mode and kernel mode. A bit, called the mode bit, is added to the hardware of the computer to indicate the current mode: kernel (0) or user (1). With the mode bit, we are able to distinguish between a task that is executed on behalf of the operating system and one that is executed on behalf of the user. When the computer system is executing on behalf of a user application, the system is in user mode. However, **when a user application requests a service from the operating system (via a system call), it must transition from user to kernel mode to fulfil the request**.

At system boot time, the hardware starts in kernel mode. The operating system is then loaded and starts user applications in user mode. **Whenever a trap or interrupt occurs, the hardware switches from user mode to kernel mode** (that is, changes the state of the mode bit to 0). Thus, whenever the operating system gains control of the computer, it is in kernel mode. The system always switches to user mode (by setting the mode bit to 1) before passing control to a user program.

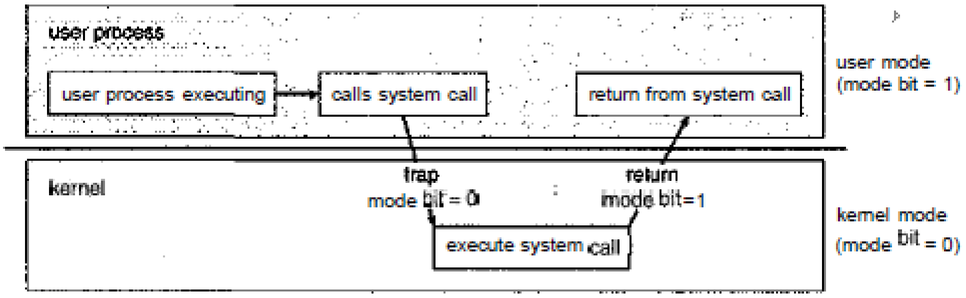


Figure 1.8 Transition from user to kernel mode.

Note: The hardware allows privileged instructions to be executed only in kernel mode. If an attempt is made to execute a privileged instruction in user mode, the hardware does not execute the instruction but rather treats it as illegal and traps it to the operating system.

The instruction to switch to user mode is an example of a privileged instruction. Some other examples include I/O control, timer management, and interrupt management.

System Call:

Control is switched back to the operating system via an interrupt, a trap, or a system call. System calls provide the means for a user program to ask the operating system to perform tasks reserved for the operating system on the user program's behalf. A system call usually takes the form of a trap to a specific location in the interrupt vector. **When a system call is executed, it is treated by the hardware as a software interrupt.**

Note: The lack of a hardware-supported dual mode can cause serious shortcomings in an operating system.

Timer:

We must ensure that the operating system maintains control over the CPU. We must prevent a user program from getting stuck in an infinite loop or not calling system services and never returning control to the operating system. To accomplish this goal, we can use a **timer**. A timer can be set to interrupt the computer after a **specified period**. The period may be fixed (for example, 1/60 second) or **variable** (for example, from 1 millisecond to 1 second). **A variable timer is generally implemented by a fixed-rate clock and a counter**. The operating system sets the counter. Every time the clock ticks, the counter is decremented. When the counter reaches 0, an interrupt occurs. Thus, we can use the timer to prevent a user program from running too long.

Process Management

A program does nothing unless its instructions are executed by a CPU. A program in execution, as mentioned, is a process. A time-shared user program such as a compiler is a process. A word-processing program being run by an individual user on a PC is a process. A system task, such as sending output to a printer, can also be a process (or at least part of one).

A process needs certain resources—including CPU time, memory, files, and I/O devices—to accomplish its task.

The operating system is responsible for the following activities in connection with process management:

- Creating and deleting both user and system processes
- Suspending and resuming processes
- providing mechanisms for process synchronization
- providing mechanisms for process communication
- providing mechanisms for deadlock handling

Memory Management

Main memory is a repository of quickly accessible data shared by the CPU and I/O devices. The central processor reads instructions from main memory during the instruction-fetch cycle and both reads and writes data from main memory during the data-fetch cycle.

For a program to be executed, **it must be mapped to absolute addresses and loaded into memory.** As the program executes, it accesses program instructions and data from memory by generating these absolute addresses. Eventually, the program terminates, its memory space is declared available, and the next program can be loaded and executed.

The operating system is responsible for the following activities in connection with memory management:

- Keeping track of which parts of memory are currently being used and by whom
- Deciding which processes (or parts thereof) and data to move into and out of memory
- Allocating and deallocating memory space as needed

Note: A batch system executes jobs, whereas a time-shared system has user programs, or tasks.

Process Management

A process is a program in execution. A process is more than the program code, which is sometimes known as the **text section**. It 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 3.1.

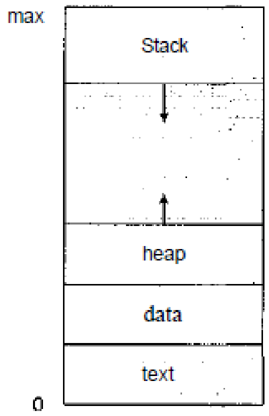


Figure 3.1 Process in memory.

A program by itself is not 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 specifying the next instruction to execute and a set of associated resources. Although two processes may be associated with the same program, **they are nevertheless considered two separate execution sequences**. For example the same user may invoke many copies of the web browser program. **Each of these is a separate process; and although the text sections are equivalent, the data, heap, and stack sections vary**.

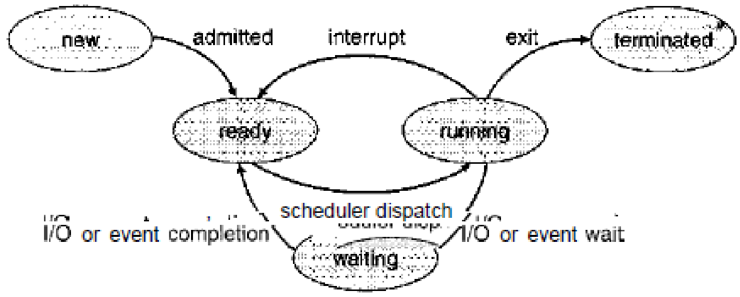


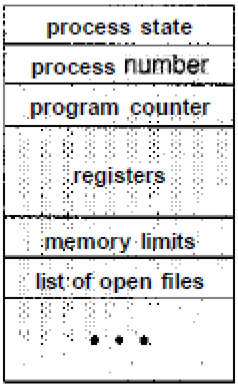
Figure 3.2 Diagram of process state.

Process State:

- **New.** The process is being created.
- **Running.** Instructions are being executed.
- **Waiting.** The process is waiting for some event to occur (such as an I/O completion or reception of a signal).
- **Ready.** The process is waiting to be assigned to a processor.
- **Terminated.** The process has finished execution.

Process Control Block:

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 3.3.



process state
process number
program counter
registers
memory limits
list of open files
• • •

Figure 3.3 Process control block (PCB).

Process state: The state may be new, ready, running, and 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. 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.
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.

Process Scheduling:

The objective of multiprogramming is to have some process running at all times, to maximize CPU utilization. The objective of time sharing is to switch the CPU among processes so frequently that users can interact with each program while it is running. To meet these objectives, the process scheduler selects an available process (possibly from a set of several available processes) for Program execution on the CPU. For a single-processor system, there will never be more than one running process. If there are more processes, the rest will have to wait until the CPU is free and can be rescheduled.

Scheduling Queues:

As processes enter the system, they are put into a job queue, which consists of all processes in the system. The processes that are residing in main memory and are ready and waiting to execute are kept on a list called the **ready queue**. This queue is generally stored as a linked list. A ready-queue header contains pointers to the first and final PCBs in the list. Each PCB includes a pointer field that points to the next PCB in the ready queue.

Suppose the process makes an I/O request to a shared device, such as a disk. Since there are many processes in the system, the disk may be busy with the I/O request of some other process. The process therefore may have to wait for the disk. The list of processes waiting for a particular I/O device is called a **device queue**. **Each device has its own device queue**.

A new process is initially put in the ready queue. It waits there until it is selected for execution, or is **dispatched**. Once the process is allocated the CPU and is executing, one of several events could occur:

- The process could issue an I/O request and then be placed in an I/O queue.
- The process could create a new subprocess and wait for the subprocess's termination.
- The process could be removed forcibly from the CPU, as a result of an interrupt, and be put back in the ready queue.

In the first two cases, the process eventually switches from the waiting state to the ready state and is then put back in the ready queue. A process s continues this cycle until it terminates, at which time it is removed from all queues and has its PCB and resources deallocated.

Schedulers

A process migrates among the various scheduling queues throughout its lifetime.

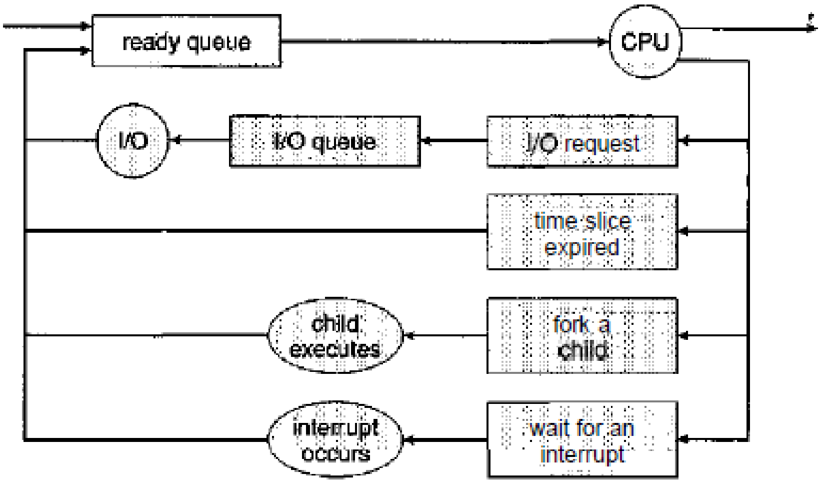


Figure 3.7 Queueing-diagram representation of process scheduling.

Long Term Scheduler:

The **long-term scheduler**, or **job scheduler**, selects processes from this pool and loads them into memory for execution.

a. The long-term scheduler executes much less frequently.
b. The long-term scheduler controls the **degree of multiprogramming** (the number of processes in memory).
c. If the degree of multiprogramming is stable, then the average rate of process creation must be equal to the average departure rate of processes leaving the system. Thus, the long-term scheduler may need to be invoked only when a process leaves the system.
d. Because of the longer interval between executions, the long-term scheduler can afford to take more time to decide which process should be selected for execution.

It is important that the long-term scheduler select a good process mix of **I/O-bound** and **CPU-bound** processes. If all processes are I/O bound, the ready queue will almost always be empty, and the short-term scheduler will have little to do. If all processes are CPU bound, the I/O waiting queue will almost always be empty, devices will go unused.
Note: On some systems, the long-term scheduler may be absent or minimal. Simply put every new process in memory for the short-term scheduler.

Short Term Scheduler:

The **short-term scheduler**, or **CPU scheduler**, selects from among the processes that are ready to execute and allocates the CPU to one of them. The short-term scheduler executes **very frequently** and should be fast to select a process.

Medium-term scheduler:

Some operating systems, such as **time-sharing systems**, may introduce an additional, intermediate level of scheduling. The key idea behind a medium-term scheduler is that sometimes it can be advantageous to remove processes from memory and **thus reduce the degree** of multiprogramming. Later, the process can be reintroduced into memory, and its execution can be continued where it left off. This scheme is called **swapping**.

Context Switch:

Interrupts cause the operating system to change a CPU from its current task and to run a kernel routine. When an interrupt occurs, the system needs to save the current context of the process currently running on the CPU so that it can restore that context when its processing is done, essentially suspending the process and then resuming it. The context is represented in the PCB of the process; switching the CPU to another process requires performing a **state save** of the current process and a **state restore** of a different process. This task is known as a **context switch**. When a context switch occurs, the kernel saves the context of the old process in its PCB and loads the saved context of the new process scheduled to run. Context-switch time is pure overhead.

Note: Context-switch times are highly dependent on hardware support.

Process Creation:

When a process creates a new process, two possibilities exist in terms of execution:
1. The parent continues to execute concurrently with its children.
2. The parent waits until some or all of its children have terminated.

There are also two possibilities in terms of the address space of the new process:
1. The child process is a duplicate of the parent process (it has the same program and data as the parent).
2. The child process has a new program loaded into it.

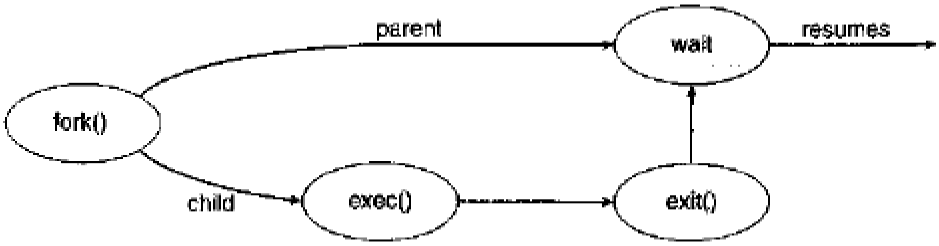


Figure 3.11 Process creation.

Process Termination:

A process terminates when it finishes executing its final statement and asks the operating system to delete it by using the **exit ()** system call.

A parent may terminate the execution of one of its children for a variety of reasons, such as these:

- The child has exceeded its usage of some of the resources that it has been allocated.
- The task assigned to the child is no longer required.
- The parent is exiting, and the operating system does not allow a child to continue if its parent terminates.

Threads

A thread is a basic unit of CPU utilization; it comprises a **thread ID, a program counter, a register set, and a stack.** It shares with other threads belonging to the same process **its code section, data section and other operating-system resources, such as open files and signals.** A **traditional** (or heavyweight) process has a single thread of control. If a process has multiple threads of control, it can perform more than one task at a time.

For example: A web browser might have one thread display images or text while another thread retrieves data from the network. A busy web server may have several (perhaps thousands) of clients concurrently accessing it. If the web server ran as a traditional single-threaded process, it would be able to service only one client at a time.

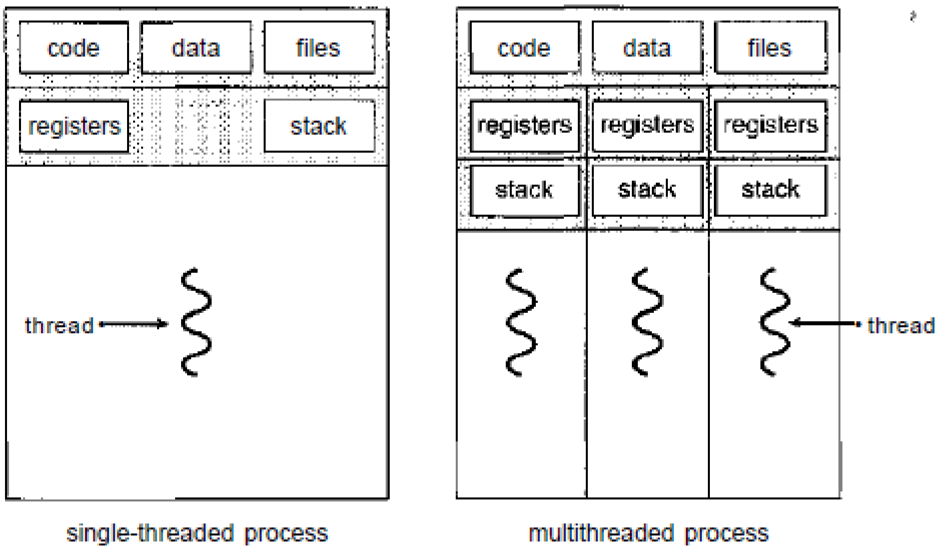


Figure 4.1 Single-threaded and multithreaded processes.

Note: Threads also play a vital role in remote procedure call (RPC) systems.

Benefits:

1. **Responsiveness:** Multithreading an interactive application may allow a program to continue running even if part of it is blocked or is performing a lengthy operation. For instance, a multithreaded web browser could still allow user interaction in one thread while an image was being loaded in another thread.
2. **Resource sharing:** By default, threads share the memory and the resources of the process to which they belong. The benefit of sharing code and data is that it allows an application to have several different threads of activity within the same address space.
3. **Economy:** Allocating memory and resources for process creation is costly. Because threads share resources of the process to which they belong, it is more **economical to create** and **context-switch threads**.
4. **Utilization of multiprocessor architectures:** A single-threaded process can only run on one CPU, no matter how many are available. Multithreading on a multi-CPU machine increases concurrency.

Multithreading Models:

Support for threads may be provided either at the user level, for user threads, or by the Kernel, for kernel threads. **User threads are supported above the kernel and are managed without kernel support**, whereas kernel threads are supported and managed directly by the operating system. There must exist a relationship between user threads and kernel threads.

Many-to-One Model: The many-to-one model maps many user-level threads to one kernel thread. Thread management is done by the thread library in user space, so it is efficient; but the entire process will block if a thread makes a blocking system call Also, because only one thread can access the kernel at a time, multiple threads are unable to run in parallel on multiprocessors.

One-to-One Model: It maps each user thread to a kernel thread. It provides more concurrency than the many-to-one model by allowing another thread to run when a thread makes a blocking system call; it also allows multiple threads to run in parallel on multiprocessors. The only drawback to this model is that creating a user thread requires creating the corresponding kernel thread. The overhead of creating kernel threads can burden the performance of an application.

Many-to-Many Model: The many-to-many model multiplexes many user-level threads to a smaller or equal number of kernel threads.

Thread Libraries:

A **thread library** provides the programmer an API for creating and managing threads. There are two primary ways of implementing a thread library. The first approach is to provide a library entirely in user space with no kernel support.

The second approach is to implement a kernel-level library supported directly by the operating system.

Summary:

A thread is a flow of control within a process. A multithreaded process contains several different flows of control within the same address space.

a. User-level threads are threads that are visible to the programmer and are unknown to the kernel.
b. The operating-system kernel supports and manages kernel-level threads.
c. user-level threads are faster to create and manage than are kernel threads, as no intervention from the kernel is required
d. Multithreaded programs introduce many challenges for the programmer, including the semantics of the fork() and exec() system calls.

CPU Scheduling

CPU scheduling is the basis of **multi-programmed** operating systems. By switching the CPU among processes, the operating system can make the computer more productive. In a single-processor system, only one process can run at a time; any others must wait until the CPU is free and can be rescheduled. The objective of multiprogramming is to have some process running at all times, to maximize CPU utilization. Every time one process has to wait, another process can take over use of the CPU.

CPU-I/O Burst Cycle:

Process execution consists of a cycle of CPU execution and I/O wait. Processes alternate between these two states. Process execution begins with a CPU burst. That is followed by an I/O burst, which is followed by another CPU burst, then another I/O burst, and so on. Eventually, the final CPU burst ends with a system request to terminate execution.

CPU Scheduler:

Whenever the CPU becomes idle, the operating system must select one of the processes in the ready queue to be executed. The selection process is carried out by the short-term scheduler (or CPU scheduler). The scheduler selects a process from the processes in memory that are ready to execute and allocates the CPU to that process.

Note: The ready queue is not necessarily a first-in, first-out (FIFO) queue. A ready queue can be implemented as a FIFO queue, a priority queue, a tree, or simply an unordered linked list.

Pre-emptive Scheduling:

CPU-scheduling decisions may take place under the following four circumstances:

1. When a process switches from the running state to the waiting state (for example, as the result of an I/O request).
2. When a process switches from the running state to the ready state (For example, when an interrupt occurs)
3. When a process switches from the waiting state to the ready state (for eg, at completion of I/O)
4. When a process terminates

For situations **1 and 4**, there is no choice in terms of scheduling. A new process (if one exists in the ready queue) must be selected for execution. We say that the scheduling scheme is **nonpreemptive** or **cooperative**. Under nonpreemptive scheduling, once the CPU has been allocated to a process, the process keeps the CPU until it releases the CPU either by terminating or by switching to the waiting state. **Note: because it does not require the special hardware (for example, a timer).**

There is a choice, however, for situations **2 and 3**. Unfortunately, **preemptive** scheduling incurs a cost associated with access to shared data. Consider the case of two processes that share data. While one is updating the data, it is preempted so that the second process can run. The second process then tries to read the data, which are in an inconsistent state.

Dispatcher:

The dispatcher is the module that **gives control of the CPU to the process** selected by the short-term scheduler. This function involves the following:

- Switching context
- Switching to user mode
- Jumping to the proper location in the user program to restart that program

The dispatcher should be as fast as possible, since it is invoked during every process switch. The time it takes for the dispatcher to stop one process and start another running is known as the **dispatch latency**.

Scheduling Criteria:

Different CPU scheduling algorithms have different properties, and the choice of a particular algorithm may favour one class of processes over another.
CPU utilization: We want to keep the CPU as busy as possible.
Throughput: If the CPU is busy executing processes, then work is being done. One measure of work is the number of processes that are completed per time unit, called throughput.
Turnaround time: The important criterion is how long it takes to execute that process. The interval from the time of submission of a process to the time of completion is the **turnaround time**. Turnaround time is the sum of the periods spent waiting to get into memory, waiting in the ready queue, executing on the CPU, and doing I/O.
Waiting time: The CPU scheduling algorithm does not affect the amount of time during which a process executes or does I/O; it affects only the amount of time that a process spends waiting in the ready queue. Waiting time is the sum of the periods spent waiting in the ready queue.
Response time: Often, a process can produce some output fairly early and can continue computing new results while previous results are being output to the user. Thus, another measure is the time from the submission of a request until the first response is produced. This measure, called **Response Time**, is the time it takes to start responding, not the time it takes to output the response.

Note:

1. **The turnaround time is generally limited by the speed of the output device.**
2. **It is desirable to maximize CPU utilization and throughput and to minimize turnaround time, waiting time, and response time.**
3. **A system with reasonable and predictable response time may be considered more desirable than a system that is faster on the average but is highly variable**

Scheduling Algorithms

First-Come, First-Served Scheduling (Non Pre-emptive)(Arrival Time)

Simplest CPU-scheduling algorithm is the **first-come, first-served**, the implementation of the FCFS policy is easily managed with a FIFO queue. When a process enters the ready queue, its PCB is linked onto the tail of the queue. When the CPU is free, it is allocated to the process at the head of the queue.
Note: average waiting time under the FCFS policy, however, is often quite long. Once the CPU has been allocated to a process, that process keeps the CPU until it releases the CPU, either by **terminating** or by **requesting I/O.**
It suffers from **convoy** effect.

Shortest-Job-First Scheduling (Non-Prepemptive) (Next Burst Time)(Starvation)

This algorithm associates with each process the length of the **process's next CPU burst**. When the CPU is available, it is assigned to the process that has the **smallest next CPU burst**. If the next CPU bursts of two processes are the same, **FCFS scheduling is used to break the tie**.
A more appropriate term for this scheduling method would be the **shortest-next-CPU-burst** algorithm, because scheduling depends on the length of the next CPU burst of a process, rather than its total length.
Note: The SJF scheduling algorithm is **provably optimal**, in that it gives the **minimum average waiting time** for a given set of processes. **SJF scheduling is used frequently in long-term scheduling**.

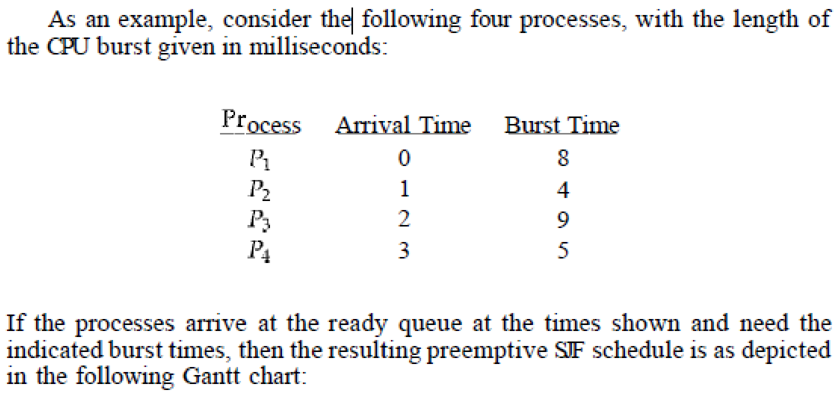
The real difficulty with the SJF algorithm is knowing the length of the next CPU request. Although the SJF algorithm is optimal, it cannot be implemented at the level of short-term CPU scheduling. There is no way to know the length of the next CPU burst. One approach is to try to approximate SJF scheduling. We may not know the length of the next CPU burst, but we may be able to predict its value.

1. $t_n =$ actual length of n^{th} CPU burst
2. $\tau_{n+1} =$ predicted value for the next CPU burst
3. $\alpha, 0 \le \alpha \le 1$
4. Define:

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

This formula defines an **exponential average**. The value of **tn** contains our most recent information; **τn**(tau n) stores the past history. The parameter **α** controls the relative weight of recent and past history in our prediction. If **α** = 0, then **τn+1 = τn**, and recent history has no effect.
. If **α** = 1, then **τn+1 = tn**, and only most recent CPU burst matters (history is assumed to be old and irrelevant). More commonly, **α** = 1/2, so recent history and past history are equally weighted.
The initial **τ0** can be defined as a constant or as an overall system average.

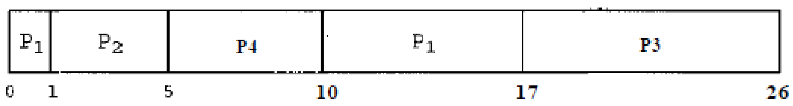
Shortest-remaining-time-first scheduling (Pre-emptive)(Starvation)



As an example, consider the following four processes, with the length of the CPU burst given in milliseconds:

Process	Arrival Time	Burst Time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

If the processes arrive at the ready queue at the times shown and need the indicated burst times, then the resulting preemptive SJF schedule is as depicted in the following Gantt chart:



Priority Scheduling (Pre-emptive)(Starvation)

An SJF algorithm is simply a priority algorithm where the priority (p) is the inverse of the (predicted) next CPU burst. The larger the CPU burst, the lower the priority, and vice versa. When a process arrives at the ready queue, its priority is compared with the priority of the currently running process.
A major problem with priority scheduling algorithms **is indefinite blocking**, or **starvation**. A solution to the problem of indefinite blockage of low-priority processes is **aging**. Aging is a technique of gradually increasing the priority of processes that wait in the system for a long time.

Round-Robin Scheduling (Pre-emptive) (No Starvation)

The **round-robin (RR)** scheduling algorithm is designed especially for timesharing systems. It is similar to FCFS scheduling, but preemption is added to switch between processes. A small unit of time, called a **time quantum** or **time slice**, is defined. A time quantum is generally from 10 to 100 milliseconds. **The ready queue is treated as a circular queue**. The CPU scheduler goes around the ready queue, allocating the CPU to each process for a time interval of up to 1 time quantum.
To implement RR scheduling, we keep the ready queue as a FIFO queue of processes. New processes are added to the tail of the ready queue. The CPU scheduler picks the first process from the ready queue, sets a timer to interrupt after 1 time quantum, and dispatches the process.

Note: The **average waiting time** under the RR policy is often long. In the RR scheduling algorithm, no process is allocated the CPU for more than 1 time quantum in a row (unless it is the only runnable process). **That a context switch doesn't take place if there is only one process (Stackoverflow)**

If there are n processes in the ready queue and the time quantum is q, then each process gets 1/n of the CPU time in chunks of at most q time units. Each process must wait no longer than (n — 1) x q time units until its next time quantum.
The performance of the RR algorithm depends heavily on the size of the time quantum. At one extreme, if the time quantum is extremely large, the RR policy is the same as the FCFS policy. If the time quantum is extremely small (say, 1 millisecond), the RR approach is called processor sharing and (in theory) creates the appearance that each of n processes has its own processor running at 1/n the speed of the real processor.

Let us assume that we have only one process of 10 time units. If the quantum is 12 time units, the process finishes in less than 1 time quantum, with no overhead. If the time quantum is 1 time unit, then nine context switches will occur, slowing the execution of the process.

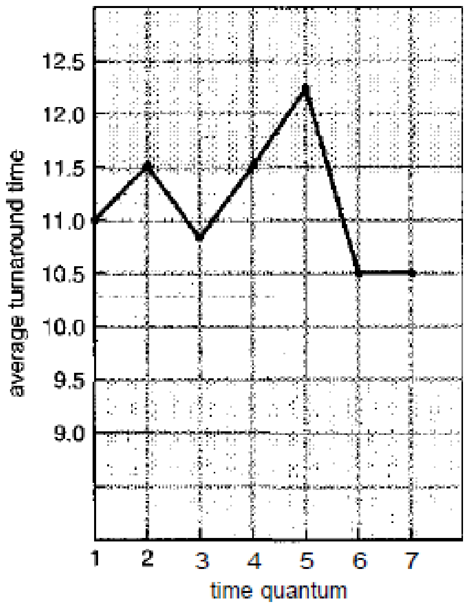


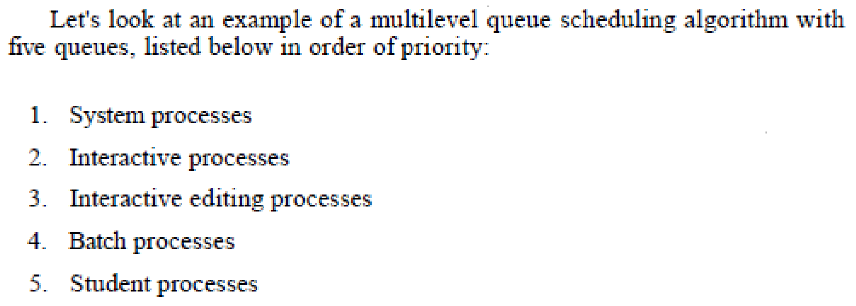
Figure 5.5 The way in which turnaround time varies

The **average turnaround time** of a set of processes **does not** necessarily improve as the time-quantum size increases. **The average turnaround time increases for a smaller time quantum, since more context switches are required**.

Multilevel Queue Scheduling

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. 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**. The foreground queue might be scheduled by an RR algorithm, while the background queue is scheduled by an FCFS algorithm. In addition, there must be scheduling among the queues, which is commonly implemented as fixed-priority pre-emptive scheduling. For example, the foreground queue may have absolute priority over the background queue.



Let's look at an example of a multilevel queue scheduling algorithm with five queues, listed below in order of priority:

1. System processes
2. Interactive processes
3. Interactive editing processes
4. Batch processes
5. Student processes

Each queue has absolute priority over lower-priority queues. No process in the batch queue, for example, could run unless the queues for system processes, interactive processes, and interactive editing processes were all empty. If an interactive editing process entered the ready queue while a batch process was running, the batch process would be pre-empted.

Multilevel Feedback-Queue Scheduling (No Starvation)

Normally, when the multilevel queue scheduling algorithm is used, processes are permanently assigned to a queue when they enter the system. If there are separate queues for foreground and background processes, for example, **processes do not move from one queue to the other**, since processes do not change their foreground or background nature.

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. 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**.

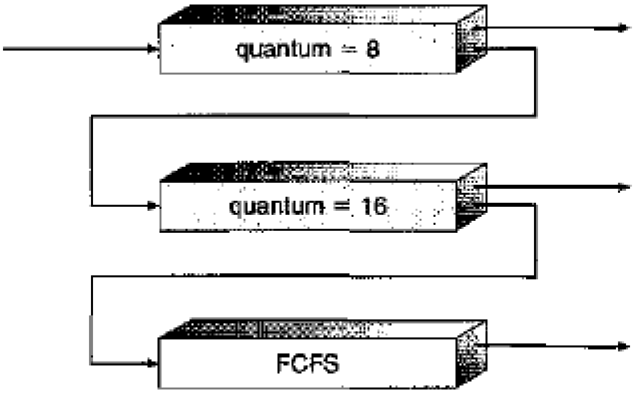
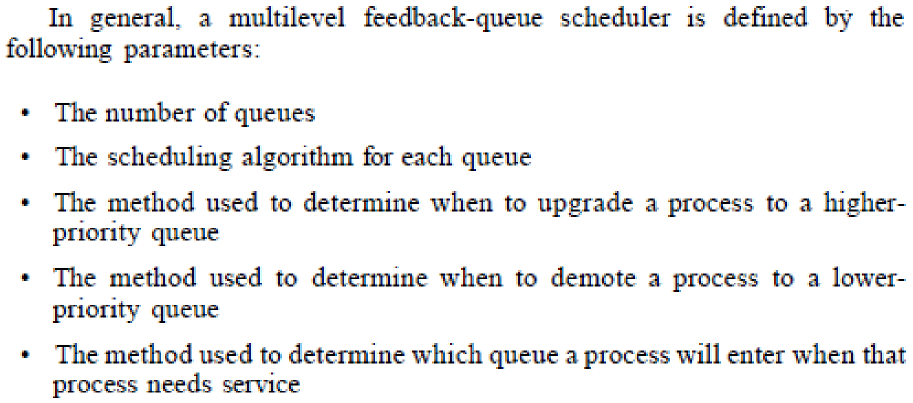


Figure 5.7 Multilevel feedback queues.

The scheduler first executes all processes in queue 0. Only when queue 0 is empty will it execute processes in queue 1. Similarly, processes in queue 2 will only be executed if queues 0 and 1 are empty. A process that arrives for queue 1 will preempt a process in queue 2. A process in queue 1 will in turn be preempted by a process arriving for queue 0.

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.



In general, a multilevel feedback-queue scheduler is defined by the following parameters:

- The number of queues
- The scheduling algorithm for each queue
- The method used to determine when to upgrade a process to a higher-priority queue
- The method used to determine when to demote a process to a lower-priority queue
- The method used to determine which queue a process will enter when that process needs service

Deadlock

In a multiprogramming environment, several processes may compete for a finite number of resources. A process requests resources; and if the resources are not available at that time, the process enters a waiting state. Sometimes, a waiting process is never again able to change state, because the resources it has requested are held by other waiting processes. This situation is called a **deadlock**.
A system consists of a finite number of resources to be distributed among a number of competing processes. A system has two CPUs, then the resource type CPU has two instances. Similarly, the resource type printer may have five instances.

Request If the request cannot be granted immediately (for example, if the resource is being used by another process), then the requesting process must wait until it can acquire the resource.
Use The process can operate on the resource (for eg: the process can print on the printer).
Release The process releases the resource.

Note: The request and release of resources are system calls. Multithreaded programs are good candidates for deadlock because multiple threads can compete for shared resources.

"A set of processes is in a deadlock state when every process in the set is waiting for an event that can be caused only by another process in the set"

Deadlock Characterization

Necessary Conditions: A deadlock situation can arise if the following four conditions hold "simultaneously" in a system:

Mutual exclusion "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.

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

No preemption 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.

Circular wait A set {P0, Pi, ..., Pn} of waiting processes must exist such that P0 is waiting for a resource held by P1, P1 is waiting for a resource held by P2, Pn-1 is waiting for a resource held by Pn, and Pn is waiting for a resource held by P0.

Resource-Allocation Graph
Deadlocks can be described more precisely in terms of a directed graph called a system resource-allocation graph.
A **directed edge** from process Pi to resource type Rj is denoted by Pi -> Rj (**Request Edge**) it signifies that process Pi has requested an instance of resource type Rj and is currently waiting for that resource.
A **directed edge** from resource Rj to process Pi (**Assignment Edge**) is denoted by Rj -> Pi it signifies that that an instance of resource type Rj has been allocated to process Pi.

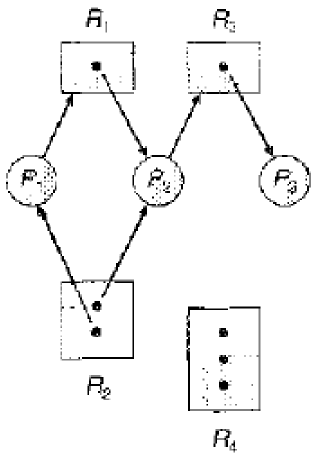


Figure 7.2 Resource-allocation graph.

"If each resource type has exactly one instance, then a cycle implies that a deadlock has occurred"

Methods for Handling Deadlocks

We can deal with the deadlock problem in one of three ways:

- We can use a protocol to prevent or avoid deadlocks, ensuring that the system will never enter a deadlock state.
- We can allow the system to enter a deadlock state, detect it, and recover.
- We can ignore the problem altogether and pretend that deadlocks never occur in the system.

The third solution is the one used by most operating systems it is then up to the application developer to write programs that handle deadlocks.

To ensure that deadlocks never occur, the system can use either a **deadlock prevention** or a **deadlock-avoidance** scheme.

Deadlock prevention provides a set of methods for ensuring that at least one of the necessary conditions cannot hold. These methods prevent deadlocks by constraining how requests for resources can be made.

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.

Detect & Recover If a system does not employ either a deadlock-prevention or a deadlock avoidance algorithm, then a deadlock situation may arise. In this environment, the system can provide an algorithm that examines the state of the system to determine whether a deadlock has occurred and an algorithm to recover from the deadlock (if a deadlock has indeed occurred).

Ignore Deadlock If a system neither ensures that a deadlock will never occur nor provides a mechanism for deadlock detection and recovery. If system enters a deadlocked state. The System will stop functioning and will need to be restarted manually. Although this method may not seem to be a viable approach to the deadlock problem, it is nevertheless used in most operating systems this method is cheaper than the prevention, avoidance, or detection and recovery methods.

Deadlock Prevention

For a deadlock to occur, each of the four necessary conditions must hold. By ensuring that at least one of these conditions cannot hold, we can prevent the occurrence of a deadlock.

Mutual exclusion the mutual-exclusion condition in deadlock must hold for non-sharable resources. For example, a printer cannot be simultaneously shared by several processes. We cannot prevent deadlocks by denying the mutual-exclusion condition, because some resources are intrinsically non-sharable (we can't make printer shareable like files)

Hold and Wait

To ensure that the hold-and-wait condition never occurs in the system, we must guarantee that:

1. Whenever a process requests a resource, it does not hold any other resources.
2. One protocol that can be used requires each process to request and be allocated all its resources before it begins execution. (resource utilization may be low, resources may be allocated but unused for a long period)
3. An alternative protocol allows a process 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 release all the resources that it is currently allocated (starvation is possible, A process that needs several resources may have to wait indefinitely, because at least one of the resources that it needs is always allocated to some other process)

No Preemption

To ensure that this condition does not hold, we can use the following protocol:

1. If a process is holding some resources and requests another resource that cannot be immediately allocated to it (that is, the process must wait), then all resources currently being held are pre-empted the preempted resources are added to the list of resources for which the process is waiting. The process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.
2. Alternatively, if a process requests some resources, we first check whether they are available. If they are, we allocate them. If they are not, we check whether they are allocated to some other process that is waiting for additional resources. If so, we preempt the desired resources from the waiting process and allocate them to the requesting process. If the resources are neither available nor held by a waiting process, the requesting process must wait. While it is waiting, some of its resources may be preempted, but only if another process requests them. A process can be restarted only when it is allocated the new resources it is requesting and recovers any resources that were pre-empted while it was waiting.

Circular Wait

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.

Deadlock Avoidance:

Deadlock-prevention algorithms prevent deadlocks by restraining how requests can be made, and causes low device utilization and reduced system throughput.
An alternative method for avoiding deadlocks is to require additional information about how resources are to be requested.
The simplest and most useful algorithm requires that each process declare the maximum number of resources of each type that it may need.
A deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that a circular wait condition can never exist.

Safe State A state is safe if the system can allocate resources to each process (up to its maximum) in some order and still avoid a deadlock. More formally, a system is in a safe state only if there exists a safe sequence. If no such sequence exists, then the system state is said to be **unsafe**.

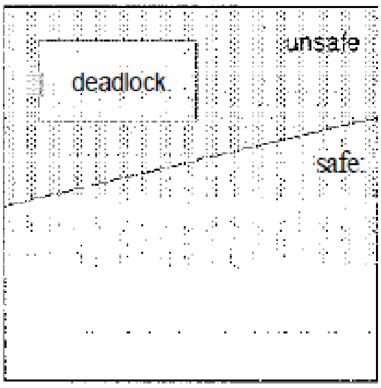


Figure 7.5 Safe, unsafe, and deadlock state spaces.

A safe state is not a deadlocked state. Conversely, **a deadlocked state is an unsafe state**. Not all unsafe states are deadlocks, however (Figure 7.5). An unsafe state may lead to a deadlock. As long as the state is safe, the operating system can avoid unsafe (and deadlocked) states.
Note: In an unsafe state, the operating system cannot prevent processes from requesting resources such that a deadlock occurs.

Whenever a process requests a resource that is currently available, the system must decide whether the resource can be allocated immediately or whether the process must wait. The request is granted only if the allocation leaves the system in a safe state.
Note: In this scheme, if a process requests a resource that is currently available, it may still have to wait. Thus, resource utilization may be lower than it would otherwise be.

Banker's Algorithm

"The resource-allocation-graph algorithm is not applicable to a resource allocation System with multiple instances of each resource type"
Banker's algorithm is applicable to system with multiple instances but is less efficient than the resource-allocation graph scheme.

When, a new process enters the system, it must declare the maximum number of instances of each resource type that it may need. This number may not exceed the total number of resources in the system. When a user requests a set of resources, the system must determine whether the allocation of these resources will leave the system in a safe state. If it will, the resources are allocated; otherwise, the process must wait until some other process releases enough resources.

Let n be the number of processes in the system and m be the number of resource types. We need the following data structures:

Available: A vector of length m indicates the number of available resources of each type. If Available[j] equals k, there are k instances of resource type Rj available.
Max: An n x m matrix defines the maximum demand of each process. If M[i][j] equals k, then process Pi may request at most k instances of resource type Rj.
Allocation: An n x in matrix defines the number of resources of each type currently allocated to each process. If Allocation[i][j] = k, then process Pi is currently allocated k instances of resource type Rj.
Need: An nxm matrix indicates the remaining resource need of each process. If Need[i][j] equals k, then process Pi may need k more instances of resource type Rj to complete its task.

Deadlock Detection

If a system does not employ either a deadlock-prevention or a deadlock avoidance algorithm, then a deadlock situation may occur. 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

Single Instance of Each Resource Type:
If all resources have only a single instance, then we can define a deadlock detection algorithm that uses a variant of the resource-allocation graph, called a **wait-for** graph.

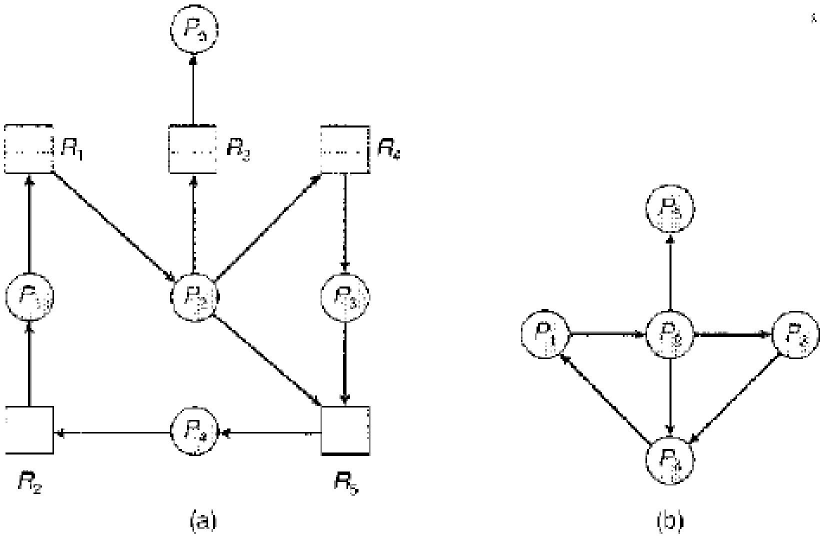


Figure 7.8 (a) Resource-allocation graph. (b) Corresponding wait-for graph.

A deadlock exists in the system if and only if the wait-for graph contains a cycle. To detect deadlocks, the system needs to maintain the wait-for graph and periodically invoke an algorithm that searches for a cycle in the graph.
"**An algorithm to detect a cycle in a graph requires an order of n1 operations, where n is the number of vertices in the graph**."

Several Instances of a Resource Type:

The wait-for graph scheme is not applicable to a resource-allocation system with multiple instances of each resource type. We turn now to a deadlock detection algorithm similar to Banker's algorithm.

Recovery from Deadlock

Process Termination

1. Abort all deadlocked processes
2. Abort one process at a time until the deadlock cycle is eliminated

Resource Preemption

To eliminate deadlocks using resource preemption, we successively pre-empt some resources from processes and give these resources to other processes until the deadlock cycle is broken.

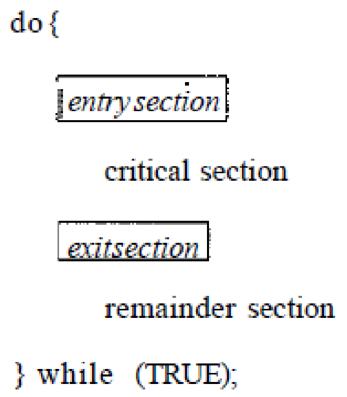
Process Synchronization

A cooperating process is one that can affect or be affected by other processes executing in the system. Cooperating processes can either directly share a logical address space (that is, both code and data) or be allowed to share data only through files or messages. The former case is achieved through the use of lightweight processes or threads.

The Critical-Section Problem

Consider a system consisting of n processes {P0, P1, ……. Pn-1}. Each process has a segment of code, called a **critical section**, in which the process may be changing common variables, updating a table, writing a file, and so on. The important feature of the system is that, when one process is executing in its critical section, no other process is to be allowed to execute in its critical section.

Note: The critical-section problem is to design a protocol that the processes can use to cooperate.



```
do {
     entry section
          critical section
     exit section
          remainder section
} while (TRUE);
```

Figure 6.1 General structure of a typical process P_i.

A solution to the critical-section problem must satisfy the following three requirements:

1. **Mutual Exclusion** If process Pi is executing in its critical section, then no other processes can be executing in their critical sections.
2. **Progress** If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in the decision on which will enter its critical section next, and this selection cannot be postponed indefinitely.
3. **Bounded waiting** there exists a bound, or limit, on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

Two general approaches are used to handle critical sections in operating systems: (1) **preemptive kernels** and (2) **nonpreemptive kernels**

A preemptive kernel allows a process to be preempted while it is running in kernel mode. A nonpreemptive kernel does not allow a process running in kernel mode to be preempted; a kernel-mode process will run until it exits kernel mode.

A nonpreemptive kernel is essentially free from race conditions on kernel data structures, as only one process is active in the kernel at a time. We cannot say the same about preemptive kernels, so they must be carefully designed to ensure that shared kernel data are free from race conditions.

Lock Variables (S/W Solution)

If Lock = 0 then C.S is free, else if Lock=1 then C.S is busy

1. Load Ri, M[Lock]
2. Comp Ri, #0
3. JNZ to step 1
4. Store M[Lock], #1
5. C.S
6. Store M[Lock], #0

Analysis:

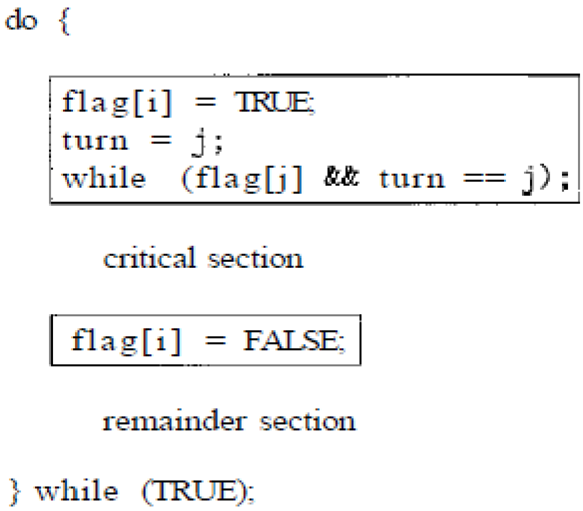
1. M.E is not satisfied, as Process P0 and P1 can execute statements 1-3 one by one and then both can enter in C.S
2. Progress is satisfied
3. Bounded Waiting is not satisfied
4. No Deadlock possible

Peterson's Solution (software-based solution)

Peterson's solution is **restricted to two processes** that alternate execution between their critical sections and remainder sections. The processes are numbered Po and P1.
For convenience, **when presenting Pi, we use Pj to denote the other process; that is, j=1-i**.

Peterson's solution requires two data items to be shared between the two processes:
int turn;
boolean flag [2].

The variable turn indicates whose turn it is to enter its critical section (if both processes are contesting for CS). That is, **if turn == i,** then process Pi is allowed to execute in its critical section. The flag array is used to indicate if a process is ready to enter its critical section. For example, if f lag[i] is true, this value indicates that Pi is ready to enter its critical section.
To enter the critical section, **process Pi, first sets flag[i] to be true and then sets turn to the value j**, thereby asserting that if the other process wishes to enter the critical section, it can do so. If both processes try to enter at the same time, turn will be set to both i and j at roughly the same time. Only one of these assignments will last; the other will occur but will be overwritten immediately. The eventual value of turn decides which of the two processes is allowed to enter its critical section first.



```
do {
     flag[i] = TRUE;
     turn = j;
     while (flag[j] && turn == j);

         critical section

     flag[i] = FALSE;

         remainder section

} while (TRUE);
```

Figure 6.2 The structure of process P_i in Peterson's solution.

We now prove that this solution is correct. We need to show that:

1. Mutual exclusion is preserved.
2. The progress requirement is satisfied.
3. The bounded-waiting requirement is met.

To prove ME, we note that each Pi enters its critical section only if either **flag[j] == false** (other process's flag is false) or **turn = i**. Also note that, if both processes can be executing in their critical sections at the same time, **then flag [0] == flag [1] == true**. These two observations imply that P0 and P1 could not have successfully executed their while statements at about the same time, since the value of turn can be either 0 or 1 but cannot be both. Hence, one of the processes say Pj must have successfully executed the while statement, whereas P, had to execute at least one additional

Statement ("turn == j"). However, since, at that time, **flag[j] == true**, and **turn == j**, and this condition will persist as long as Pj is in its critical section, the result follows: **Mutual exclusion is preserved**.

My Own Observations for M.E

1. Suppose first P0 wants to execute, it will set flag[0]=”true” and turn=1 as turn=1-0, for the process p0 the value of j is 1.
2. Now process p1 comes and sets flag[1]=”true” and sets turn=0 as for the process p1 the value of j is 0.
3. “turn” is shared among both processes while j is a private variable.
4. Now both flags are set and the current value of turn is 0, as process p1 executed later. As the value of turn is 0, then process p0 should be allowed to go to C.S, hence, first condition of while loop is correct as flag[1]=true, now second condition is checked by P0, turn=j, the value of j is 1 for P0, while the current value of turn is 0. Hence p0 enters its critical section but p1 can’t because the current value of turn is 0.

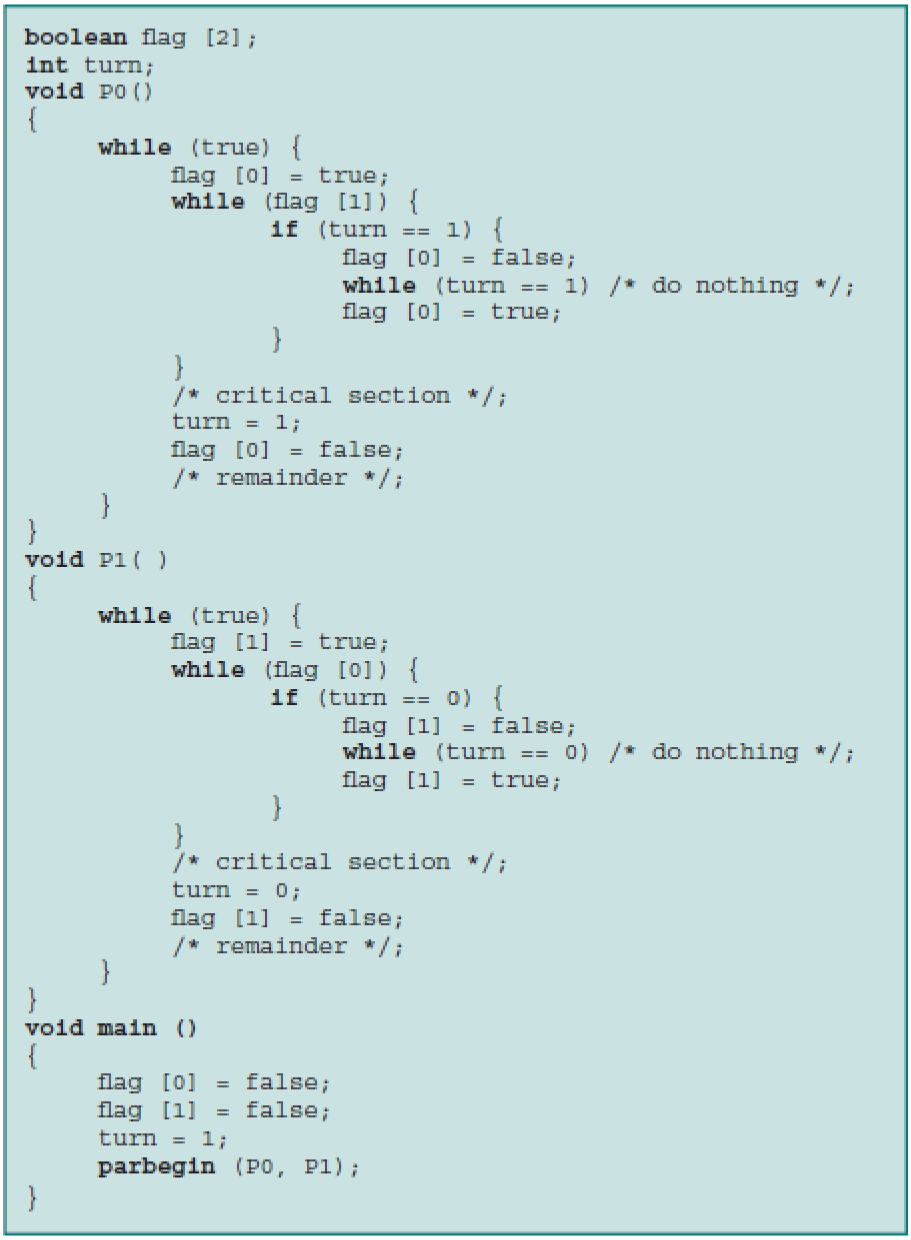
To prove properties 2 and 3, we note that a process P, can be prevented from entering the critical section only **if it is stuck in the while loop with the condition flag [j] == true and turn == j**; this loop is the only one possible. If Pj is not ready to enter the critical section, then flag [j] == false, and Pi can enter its critical section. If Pj has set flag [j] to true and is also executing in its while statement, then either turn == i or turn == j . If turn == i, then Pi will enter the critical section. If turn == j, then Pj will enter the critical section. However, once Pj exits its critical section, it will reset flag[j] to false, allowing Pi to enter its critical section. **If Pj resets flag [j] to true, it must also set turn to i**. Thus, since Pi does not change the value of the variable turn while executing the while statement, Pi will enter the critical section (**progress**) after at most one entry by Pj (**bounded waiting**).

Note: The critical-section problem could be solved simply in a uniprocessor environment if we could prevent interrupts from occurring while a shared variable was being modified.

Unfortunately, this solution is not as feasible in a multiprocessor environment. Disabling interrupts on a multiprocessor can be time consuming as the message is passed to all the processors.

<u>Dekker’s Algorithm (S/W Based Solution)</u>

The first known correct software solution to the critical-section problem for two processes was developed by Dekker



```
boolean flag [2];
int turn;
void P0()
{
     while (true) {
          flag [0] = true;
          while (flag [1]) {
               if (turn == 1) {
                    flag [0] = false;
                    while (turn == 1) /* do nothing */;
                    flag [0] = true;
               }
          }
          /* critical section */;
          turn = 1;
          flag [0] = false;
          /* remainder */;
     }
}
void P1( )
{
     while (true) {
          flag [1] = true;
          while (flag [0]) {
               if (turn == 0) {
                    flag [1] = false;
                    while (turn == 0) /* do nothing */;
                    flag [1] = true;
               }
          }
          /* critical section */;
          turn = 0;
          flag [1] = false;
          /* remainder */;
     }
}
void main ()
{
     flag [0] = false;
     flag [1] = false;
     turn = 1;
     parbegin (P0, P1);
}
```

Figure A.2 Dekker's Algorithm

Strict Alternation (only for two processes) (S/W Solution)

Process P0 Code

```
While(true)
{
Non_CS();
While(turn!=0);
Critical Section;
Turn = 1;
}
```

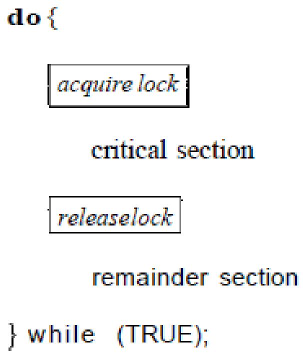
Process P1 Code

```
While(true)
{
Non_CS();
While(turn!=1);
Critical Section;
Turn = 1;
}
```

Analysis:

1. M.E is satisfied
2. Bounded waiting is satisfied, strict alternation
3. Deadlock is not possible
4. Progress is not satisfied, because if any process wants to go again into critical section can't go another process stops it.

Synchronization Hardware

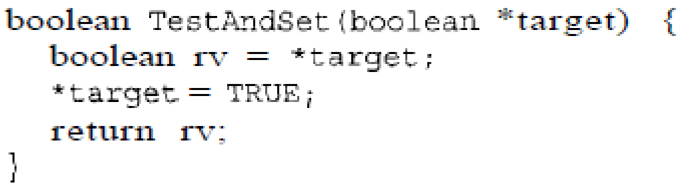


```
do {
    acquire lock
        critical section
    releaselock
        remainder section
} while  (TRUE);
```

Figure 6.3 Solution to the critical-section problem using locks.

Many modern computer systems therefore provide special hardware instructions that allow us either to test and modify the content of a word or to swap the contents of two words **atomically** that is, as one **uninterruptible** unit.

The **TestAndSet()** instruction can be defined as shown below:

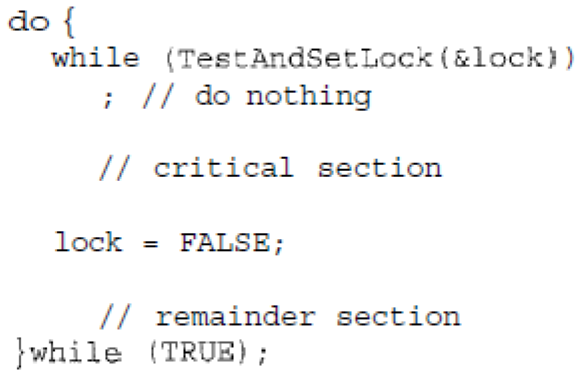


```
boolean TestAndSet(boolean *target)  {
   boolean rv = *target;
   *target = TRUE;
   return rv;
}
```

Figure 6.4 The definition of the TestAndSet() instruction.

The important characteristic is that this instruction is executed atomically. Thus, if two TestAndSet C) instructions are executed simultaneously (each on a different CPU), they will be executed sequentially in some arbitrary order. If the machine supports the TestAndSet () instruction, then we can implement mutual exclusion by declaring a Boolean variable lock, initialized to false.

The structure of process Pi is shown in figure below:



```
do {
   while (TestAndSetLock(&lock))
      ; // do nothing

      // critical section

   lock = FALSE;

      // remainder section
}while (TRUE);
```

Figure 6.5 Mutual-exclusion implementation with TestAndSet().

Note: TestAndSet() satisfies M.E and Progress but doesn't not satisfy the bounded-waiting requirement .

<u>Semaphores (O.S Solutions)</u>

A semaphore S is an integer variable that, apart from initialization, is accessed only through two standard atomic operations: **wait()** and **signal()**.
All the modifications to the integer value of the semaphore in the wait() and signal() operations must be executed **indivisibly**. That is, when one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value.

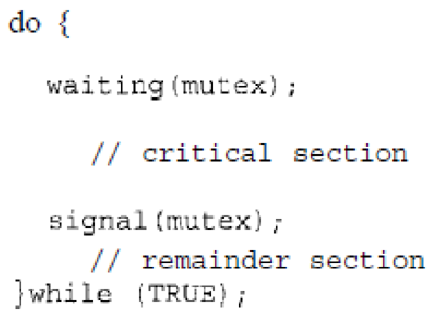
The semaphore performs 2 operations:

1. Down() or Wait() or P()
2. UP() or Signal() or V() or Release()

The value of a **counting semaphore** can range over an unrestricted domain **(- infinity to + inifinity)**. The value of a **binary semaphore** can range only between **0 and 1**. On some systems, binary semaphores are known as **mutex locks**, as they are locks that provide mutual exclusion.

We can use <u>binary semaphores to deal with the critical-section problem</u> for multiple processes. The n processes share a semaphore, mutex, initialized to 1.

<u>Counting semaphores can be used to control access to a given resource</u> consisting of a finite number of instances. The semaphore is initialized to the number of resources available. Each process that wishes to use a resource performs a wait() operation on the semaphore (decrementing the count). When a process releases a resource, it performs a signal() operation (incrementing the count). When the count for the semaphore goes to 0, all resources are being used. After that, processes that wish to use a resource will block until the count becomes greater than 0.



```
do {

  waiting(mutex);

     // critical section

  signal(mutex);
     // remainder section
}while (TRUE);
```

Figure 6.9 Mutual-exclusion implementation with semaphores.

The definition of wait() is as follows:

```
Wait(S)
  {
   While(S<=0);  //no-op
   S--;
  }
```

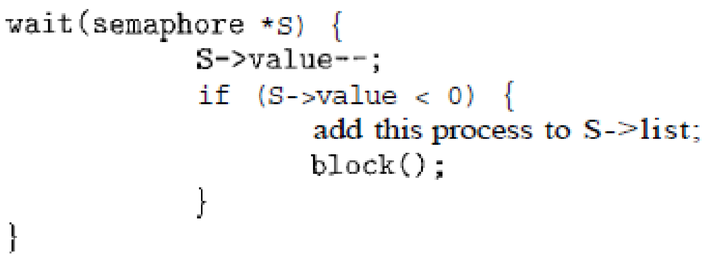
The definition of signal() is as follows:

```
Wait(S)
  {
       S++;
  }
```

The main disadvantage of the semaphore definition given here is that it requires **busy waiting**. While a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the entry code. This continual looping is clearly a problem in real multiprogramming system, where a single CPU is shared among many processes. Busy waiting wastes CPU cycles that some other process might be able to use productively. This type of semaphore is also called a **spinlock** because the process "**spins**" while waiting for the lock.

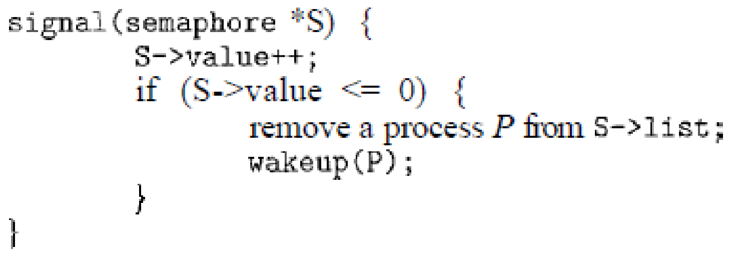
Note: Spinlocks do have an advantage in that no context switch is required when a process must wait on a lock, and a context switch may take considerable time. Thus, when locks are expected to be held for short times, spinlocks are useful; they are often employed on multiprocessor systems where one thread can "spin" on one processor while another thread performs its critical section on another processor.

To overcome the need for busy waiting, we can modify the definition of the wait() and signal() semaphore operations. Rather than engaging in busy waiting, the process can block itself. The block operation places a process into a waiting queue associated with the semaphore, and the state of the process is switched to the waiting state. Then control is transferred to the CPU scheduler, which selects another process to execute.
A process that is blocked, waiting on a semaphore S, should be restarted when some other process executes a signal() operation. The process is restarted by a wakeup() operation, which changes the process from the waiting state to the ready state. The process is then placed in the ready queue. (The CPU may or may not be switched from the running process to the newly ready process, depending on the CPU-scheduling algorithm.)
When a process must wait on a semaphore, it is added to the list of processes. A signal() operation removes one process from the list of waiting processes and awakens that process.
The wait() semaphore operation can now be defined as:



```
wait(semaphore *S) {
        S->value--;
        if (S->value < 0) {
            add this process to S->list;
            block();
        }
}
```

The signal() semaphore operation can now be defined as



```
signal(semaphore *S) {
        S->value++;
        if (S->value <= 0) {
            remove a process P from S->list;
            wakeup(P);
        }
}
```

The **block()** operation suspends the process that invokes it. The **wakeup(P)** operation resumes the execution of a blocked process P. These two operations are provided by the operating system as basic system calls.

Note: If the semaphore value is negative, its magnitude is the number of processes waiting on that semaphore.

Binary Semaphore:

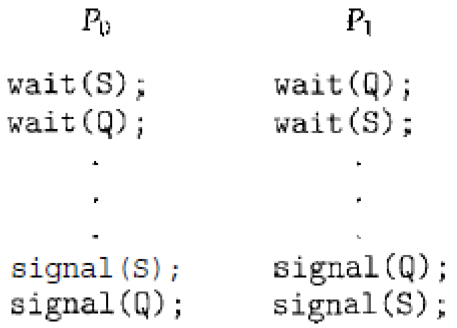
Down(Semaphore s)
{
If(s.value==1)
s.value=0;
else
{
Block the process and place its PCB in the suspend list() of S
}
}

UP(Semaphore S)
{
If(S.suspend list() is empty)
S.value=1;
else
{
Select a process from suspend list and wakeup()
}
}

Note: **Counting semaphore can be implemented by Binary Semaphore.**

Deadlocks and Starvation:

The implementation of a semaphore with a waiting queue may result in a situation where two or more processes are waiting indefinitely for an event that can be caused only by one of the waiting processes.



```
    P0              P1

wait(S);        wait(Q);
wait(Q);        wait(S);
   .               .
   .               .
   .               .
signal(S);      signal(Q);
signal(Q);      signal(S);
```

Another problem related to deadlocks is **indefinite blocking**, or **starvation**, a situation in which processes wait indefinitely within the semaphore. Indefinite blocking may occur if we add and remove processes from the list associated with a semaphore in LIFO (last-in, first-out) order.

Classic Problems of Synchronization

Producer – Consumer Problem: (Inconsistency of data)

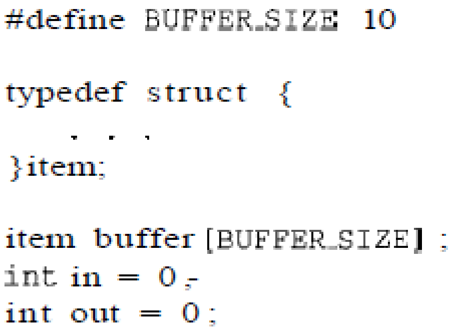
A producer process produces information that is consumed by a consumer process. For example, a compiler may produce assembly code, which is consumed by an assembler. To allow producer and consumer processes to run concurrently, we must have available a buffer of items that can be filled by the producer and emptied by the consumer. This buffer will reside in a region of memory that is shared by the producer and consumer processes. A producer can produce one item while the consumer is consuming another item.

The producer and consumer must be synchronized, so that the consumer does not try to consume an item that has not yet been produced.

Two types of buffers can be used. The **unbounded buffer** places no practical limit on the size of the buffer. The consumer may have to wait for new items, but the producer can always produce new items. The **bounded buffer** assumes a fixed buffer size. In this case, the consumer must wait if the buffer is empty, and the producer must wait if the buffer is full.

Let's look more closely at how the bounded buffer can be used to enable processes to share memory

The following variables reside in a region of memory shared by the producer & consumer processes:

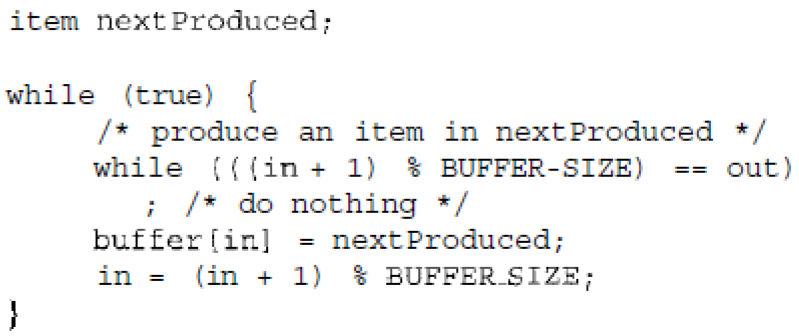


```
#define BUFFER_SIZE 10

typedef struct {
   . . .
}item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

The shared buffer is implemented as a **circular array** with two logical pointers: **in and out**. The variable **in points to the next free position in the buffer**; **out points to the first full position** in the buffer.

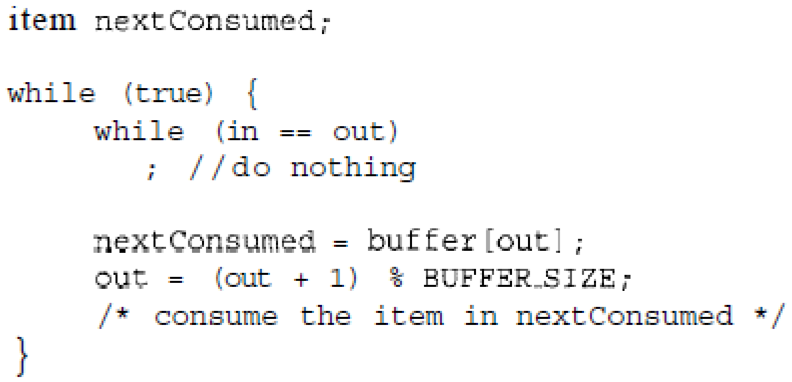
The buffer is empty when **in == out** (initially both are assigned to 0, in=0 and out=0, both are equal hence buffer is empty, when one element is inserted then in=1 and out=0,hence consumer can consume the item at 0th position); the buffer is full when **((in + 1) % BUFFER_SIZE) == out** (because of this condition producer can user only n-1 positions of buffer).

The producer process has a local variable **nextProduced** in which the new item to be produced is stored. The consumer process has a local variable **nextConsumed** in which the item to be consumed is stored.



```
item nextProduced;

while (true) {
     /* produce an item in nextProduced */
     while (((in + 1) % BUFFER-SIZE) == out)
        ; /* do nothing */
     buffer[in] = nextProduced;
     in = (in + 1) % BUFFER_SIZE;
}
```

Figure 3.14 The producer process.

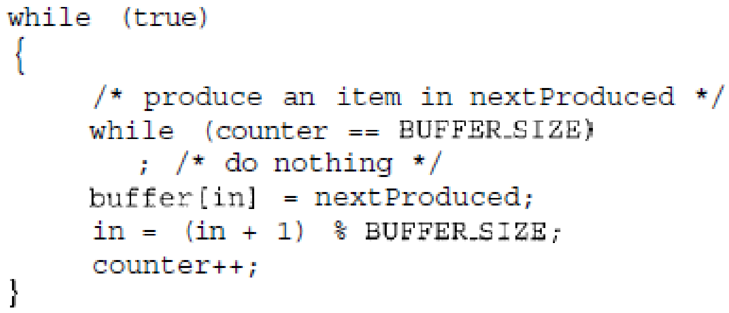


```
item nextConsumed;

while (true) {
     while (in == out)
        ; //do nothing

     nextConsumed = buffer[out];
     out = (out + 1) % BUFFER_SIZE;
     /* consume the item in nextConsumed */
 }
```

Figure 3.15 The consumer process.

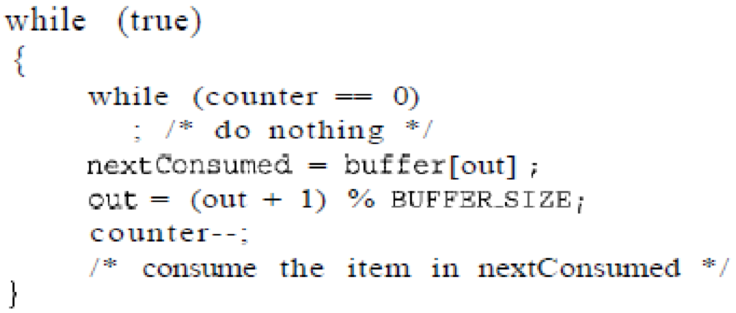
Note: It allows at most BUFFER [SIZE – 1] items in the buffer at the same time

Suppose we want to modify the algorithm to remedy this deficiency, one possibility is to add an integer variable **counter**, initialized to 0. Counter is incremented every time we add a new item to the buffer and is decremented every time we remove one item from the buffer. The code for the producer process can be modified as follows:



```
while (true)
{
     /* produce an item in nextProduced */
     while (counter == BUFFER_SIZE)
        ; /* do nothing */
     buffer[in] = nextProduced;
     in = (in + 1) % BUFFER_SIZE;
     counter++;
}
```

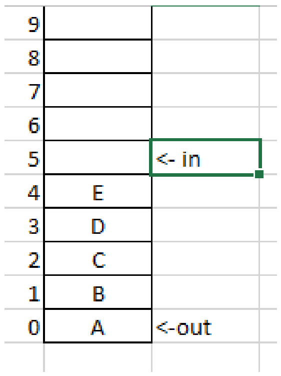
The code for the consumer process can be modified as follows:



```
while  (true)
{
      while (counter == 0)
         ; /* do nothing */
      nextConsumed = buffer[out] ;
      out = (out + 1) % BUFFER_SIZE;
      counter--;
      /* consume the item in nextConsumed */
}
```

Note: Counter and Buffer are shared resources here among producer and consumer.

Although both the producer and consumer routines are correct separately, they may not function correctly when executed concurrently

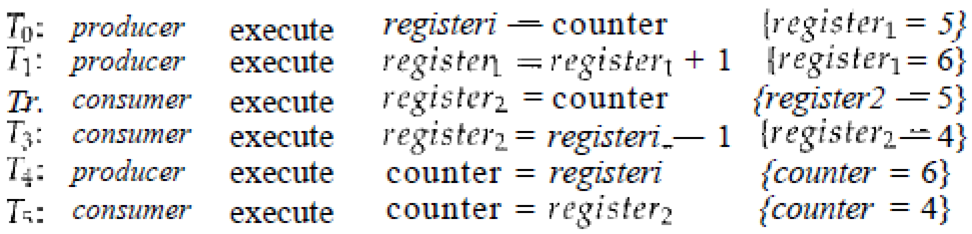


9		
8		
7		
6		
5		<- in
4	E	
3	D	
2	C	
1	B	
0	A	<-out

Value of the variable **counter** is currently 5 and that the producer and consumer processes execute the statements "**counter++**" and "**counter--**" concurrently.

"counter++" may be implemented in machine language (on a typical machine) as
register1 = counter;
register1 = register1 +1;
Counter = register1;

Similarly, the statement "counter--" is implemented as follows:
register2 = counter;
register2=counter-1;
counter= register2;
register1 and register2 are local registers.

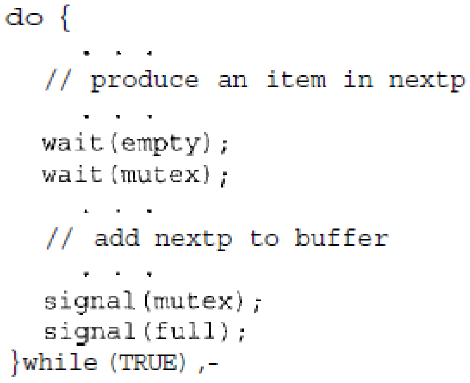


T_0:	*producer*	execute	$registeri$ — counter	$\{register_1 = 5\}$
T_1:	*producer*	execute	$registen_1 = register_1 + 1$	$\{register_1 = 6\}$
$Tr.$	*consumer*	execute	$register_2$ = counter	$\{register2 = 5\}$
T_3:	*consumer*	execute	$register_2 = registeri\_ — 1$	$\{register_2 \doteq 4\}$
T_4:	*producer*	execute	counter = $registeri$	$\{counter = 6\}$
T_5:	*consumer*	execute	counter = $register_2$	$\{counter = 4\}$

Notice that we have arrived at the incorrect state "counter == 4", indicating that four buffers are full, when, in fact, five buffers are full. If we reversed the order of the statements at T4 and T5, we would arrive at the incorrect state "counter==6".

A situation like this, where several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place, is called a race condition. To guard against the **race condition** above, we need to ensure that only one process at a time can be manipulating the variable counter. To make such a guarantee, we require that the processes be synchronized in some way.

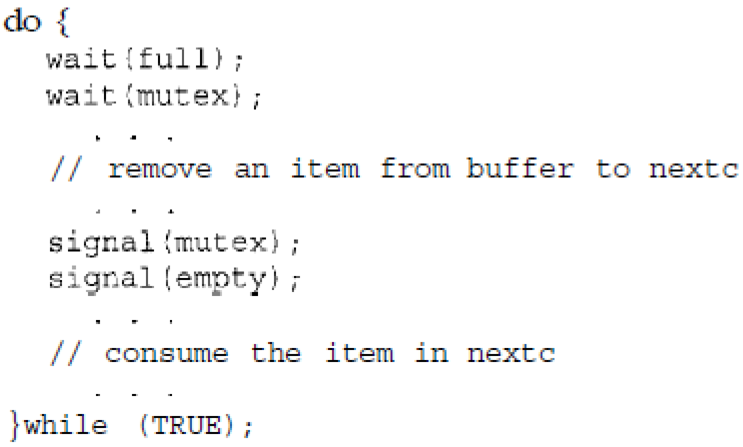
We use semaphores for synchronization:



```
do {
      . . .
   // produce an item in nextp
      . . .
   wait(empty);
   wait(mutex);
      . . .
   // add nextp to buffer
      . . .
   signal(mutex);
   signal(full);
}while (TRUE) ,-
```

Figure 6.10 The structure of the producer process.

We assume that the pool consists of n buffers, each capable of holding one item. Initially semaphore variables are assigned as
Mutex = 1; (Binary Semaphore, provides mutual exclusion)
Empty = n; (Counting Semaphore, number of empty slots)
Full = 0; (counting semaphore, number of filled slots)



```
do {
   wait(full);
   wait(mutex);
      . . .
   // remove an item from buffer to nextc
      . . .
   signal(mutex);
   signal(empty);
      . . .
   // consume the item in nextc
      . . .
}while (TRUE);
```

Figure 6.11 The structure of the consumer process.

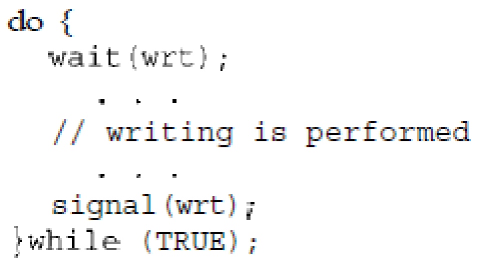
My analysis:

1. If number of empty slots are 0, then producer process will be blocked.
2. If number of filled slots are 0, then consumer process will be blocked
3. Mutex will allow only one process to execute at one time. So **"Counter"** will be either increased or decreased one at a time.
4. If **wait(empty) and Wait(mutex)** are exchanged in producer process, then if the buffer is completely filled then consumer process can't consume any item from buffer as mutex is locked by producer process.

The Readers-Writers Problem:

A database is to be shared among several concurrent processes. Some of these processes may want only to read the database, whereas others may want to update (that is, to read and write) the database. If two readers access the shared data simultaneously, no adverse effects will result. However, if a writer and some other thread (either a reader or a writer) access the database simultaneously, chaos may ensue. This synchronization problem is referred to as the **readers-writers** problem.
In the solution to the first readers-writers problem, the reader processes share the following data structures:
semaphore mutex, wrt; //Binary Semaphore
int readcount;

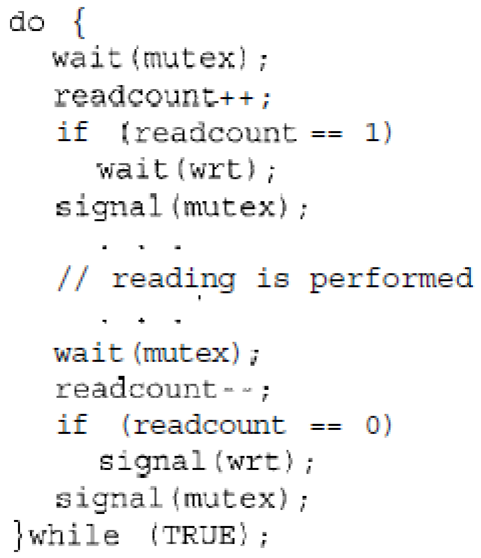
Initially, the semaphore **mutex and wrt are initialized to 1**. And **readcount is initialized to 0**.
The semaphore **wrt is common to both reader and writer processes**. **The mutex semaphore is used to ensure mutual exclusion when the variable readcount is updated.** The readcount variable keeps track of how many processes are currently reading the object. The semaphore wrt functions as a mutual-exclusion semaphore for the writers.



```
do {
   wait(wrt);
      . . .
   // writing is performed
      . . .
   signal(wrt);
}while (TRUE);
```

Figure 6.12 The structure of a writer process.

writer process will execute down operation on wrt, and will perform write operation. Until one writer finish writing till then no other write can come inside C.S and start writing because wrt is down.



```
do {
   wait(mutex);
   readcount++;
   if (readcount == 1)
      wait(wrt);
   signal(mutex);
      . . .
   // reading is performed
      . . .
   wait(mutex);
   readcount--;
   if (readcount == 0)
      signal(wrt);
   signal(mutex);
}while (TRUE);
```

Figure 6.13 The structure of a reader process.

Suppose a reader R0 wants to read, it executes the code. It performs down operation on mutex, increase readcount by 1. If readcount==1 then semaphore wrt is down, it means no writer can come inside its C.S. and it performs UP operation so that other readers can come inside C.S to perform read operation.

My Analysis:

Suppose there is only one reader, and **wait(mutex) is removed** from reader's code.

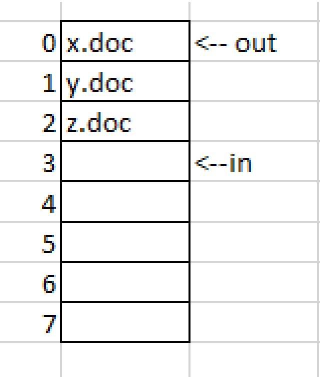
1. Process p1 goes to read and make readercount = 1 and wrt=0
2. Process p2 reads the value of readcount as 1
3. Process p1 decrements readcount and makes readcount=0, if condition is true
4. Process p1 performs UP operation on wrt
5. Process p2 makes readcount=2, because it read the value of readcount before P1 could store in memory as 1.
6. Now process p2 is reading and wrt is up hence a writer can also come in its CS along with a reader.

<u>Printer Spooler Daemon</u>: (Loss of Data)

in: variable used by all processes, in order to identify the next empty slot in spooler directory.
out: used by printer to identify from which slot it has to print a document.

Code:

1. **LOAD Ri, M[in]**
2. **STORE RD[Ri], <filename>**
3. **INCR Ri**
4. **STORE M[in], Ri**



0	x.doc	<-- out
1	y.doc	
2	z.doc	
3		<--in
4		
5		
6		
7		

Analysis:
Ready Queue: P1, P2
P1 - i R1=3
P1 - ii x.doc
P1 - iii R1=4 (not updated in memory)

P2 - i R2=3
P2 - ii y.doc (replace x.doc with y.doc at 3 cell)
P2 - iii R1=4
P1 – iv in=4
P2 – iv in=4
Note: Information has been lost about document x.doc, in is shared two process and no

<u>The Dining-Philosophers Problem</u>

Consider five philosophers who spend their lives thinking and eating. The philosophers share a circular table surrounded by five chairs, each belonging to one philosopher.

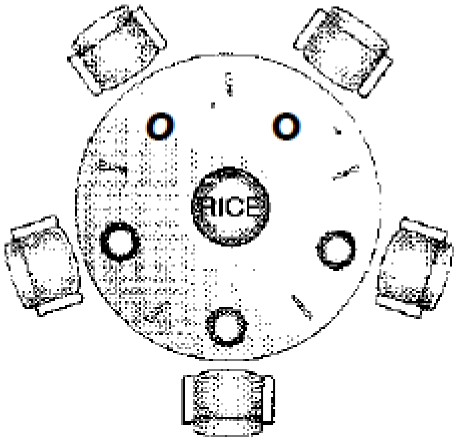


Figure 6.14 The situation of the dining philosophers.

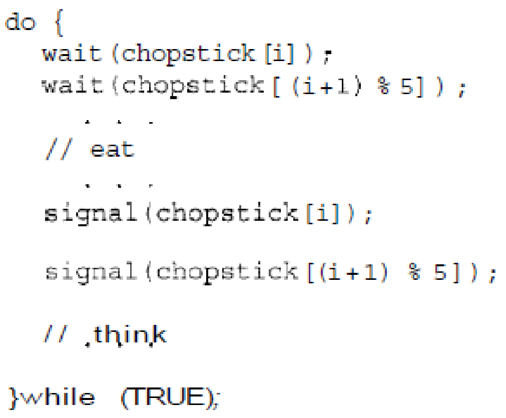
The table is laid with five single chopsticks. From time to time, a philosopher gets hungry and tries to pick up the two chopsticks that are closest to her her left and right forks, one at a time, in either order. A philosopher may pick up only one chopstick at a time. Obviously, she cannot pick up a chopstick that is already in the hand of a neighbor. When a hungry philosopher has both her chopsticks at the same time, she eats without releasing her chopsticks. When she is finished eating, she puts down both of her chopsticks and starts thinking again.

Dining-philosophers problem is a simple representation of the need to allocate several resources among several processes in a deadlock-free and starvation-free manner.

One simple solution is to represent each chopstick with a semaphore. A philosopher tries to grab a chopstick by executing a wait () operation on that semaphore; she releases her chopsticks by executing the signal() operation on the appropriate semaphores. Thus, the shared data are

semaphore chopstick[5];

Where all the elements of chopstick are initialized to 1. The structure of philosopher i is shown in below figure:



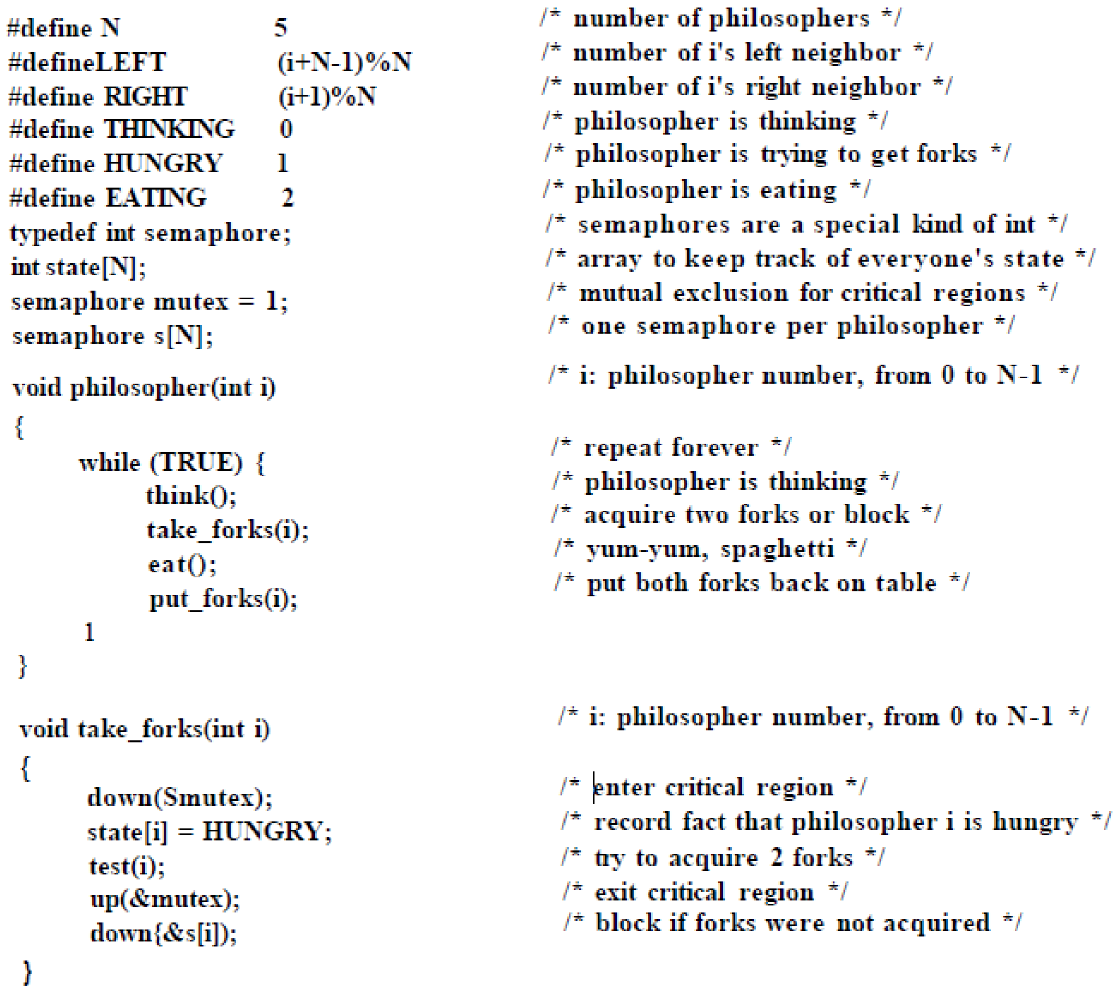
```
do {
   wait(chopstick[i]);
   wait(chopstick[(i+1) % 5]);
      . . .
   // eat
      . . .
   signal(chopstick[i]);

   signal(chopstick[(i+1) % 5]);

   // think

}while (TRUE);
```

Figure 6.15 The structure of philosopher *i*.

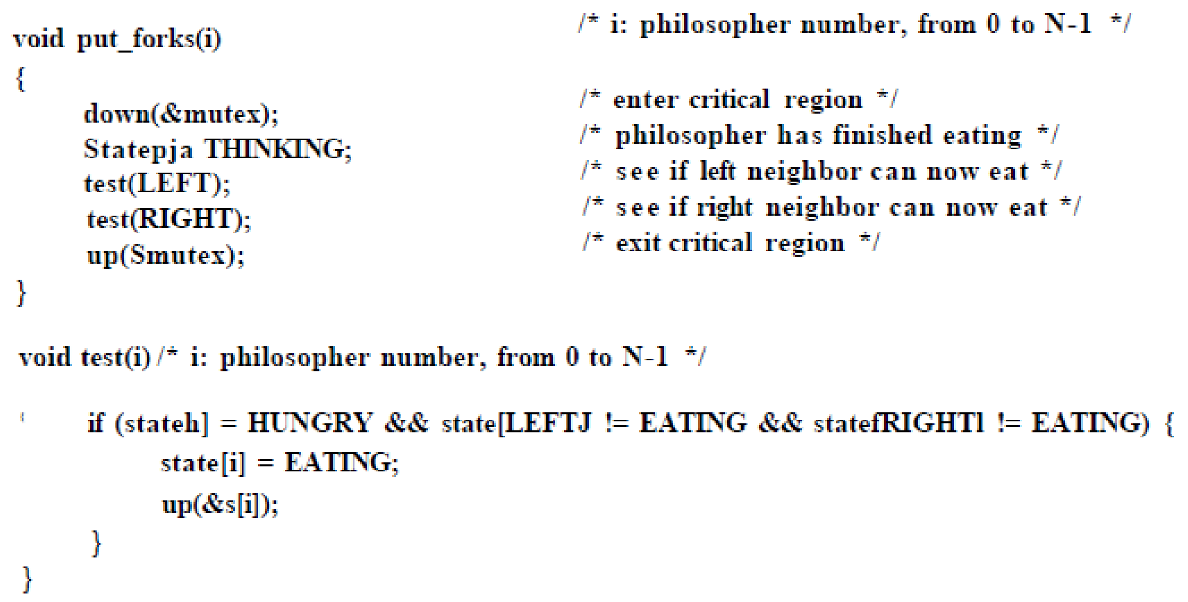
Although this solution guarantees that no two neighbours are eating simultaneously**, it must be rejected because it could create a deadlock**. Suppose that all five philosophers become hungry simultaneously and each grabs her left chopstick. All the elements of chopstick will now be equal to 0. When each philosopher tries to grab her right chopstick, she will be delayed forever.

Several possible remedies to the deadlock problem are listed next:

1. Allow at most four philosophers to be sitting simultaneously at the table
2. Allow a philosopher to pick up her chopsticks only if both chopsticks are available (to do this she must pick them up in a critical section).
3. Use an asymmetric solution; that is, an odd philosopher picks up first her left chopstick and then her right chopstick, whereas an even philosopher picks up her right chopstick and then her left chopstick.



```
#define N           5             /* number of philosophers */
#defineLEFT         (i+N-1)%N     /* number of i's left neighbor */
#define RIGHT       (i+1)%N       /* number of i's right neighbor */
#define THINKING    0             /* philosopher is thinking */
#define HUNGRY      1             /* philosopher is trying to get forks */
#define EATING      2             /* philosopher is eating */
typedef int semaphore;            /* semaphores are a special kind of int */
int state[N];                     /* array to keep track of everyone's state */
semaphore mutex = 1;              /* mutual exclusion for critical regions */
semaphore s[N];                   /* one semaphore per philosopher */

void philosopher(int i)           /* i: philosopher number, from 0 to N-1 */
{
    while (TRUE) {                /* repeat forever */
        think();                  /* philosopher is thinking */
        take_forks(i);            /* acquire two forks or block */
        eat();                    /* yum-yum, spaghetti */
        put_forks(i);             /* put both forks back on table */
    1
}

void take_forks(int i)            /* i: philosopher number, from 0 to N-1 */
{
    down(Smutex);                 /* enter critical region */
    state[i] = HUNGRY;            /* record fact that philosopher i is hungry */
    test(i);                      /* try to acquire 2 forks */
    up(&mutex);                   /* exit critical region */
    down{&s[i]);                  /* block if forks were not acquired */
}
```



```
void put_forks(i)                          /* i: philosopher number, from 0 to N-1  */
{
        down(&mutex);                      /* enter critical region */
        Statepja THINKING;                 /* philosopher has finished eating */
        test(LEFT);                        /* see if left neighbor can now eat */
        test(RIGHT);                       /* see if right neighbor can now eat */
        up(Smutex);                        /* exit critical region */
}

void test(i) /* i: philosopher number, from 0 to N-1 */

'       if (stateh] = HUNGRY && state[LEFTJ != EATING && statefRIGHTl != EATING) {
                state[i] = EATING;
                up(&s[i]);
        }
}
```

Figure 2-46. A solution to the dining philosophers problem.

This solution 2-46 is deadlock free but not starvation free. A deadlock-free solution does not necessarily eliminate the possibility of starvation.

Note:

1. If progress is **not** satisfied, starvation is possible. Eg: Strict alternation
2. Test & Set primitives are as powerful as semaphore.
3. Multiple process synchronization to a critical section can be implemented using an array of semaphores but not Binary Semaphores.
4. **Split Binary Semaphore** is essentially an array of binary semaphores.
5. A counting semaphore can be implemented using Binary Semaphore
6. **Swap Space** is the area on a hard disk which is a part of the virtual memory of your machine, which is a combination of accessible physical memory (RAM) and the swap space temporarily holds memory page that inactive when RAM is full.

Memory Management

Main memory and the registers built into the processor itself are the only storage that the CPU can access directly. There are machine instructions that take memory addresses as arguments, but none that take disk addresses. Therefore, any instructions in execution, and any data being used by the instructions, must be in one of these direct-access storage devices. If the data are not in memory, they must be moved there before the CPU can operate on them.
Registers that are built into the CPU are generally accessible within one cycle of the CPU clock. Memory access may take many cycles of the CPU clock to complete, in which case the processor normally needs to stall, since it does not have the data required to complete the instruction that it is executing.

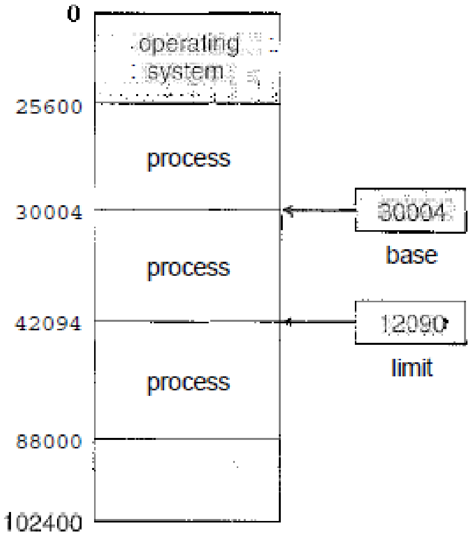


Figure 8.1 A base and a limit register define a logical address space.

Not only are we concerned with the relative speed of accessing physical memory, but we also must ensure **to protect user processes from one another**.
We first need to make sure that each process has a separate memory space. To do this, we need the ability to determine the range of legal addresses that the process may access and to ensure that the process can access only these legal addresses. We can provide this protection by using two registers, usually a base and a limit, as illustrated in Figure 8.1. The base register holds the smallest legal physical memory address; the limit register specifies the size of the range. For example**, if the base register holds 300040 and limit register is 120900, then the program can legally access all addresses from 300040 through 420940 (inclusive)**.
Protection of memory space is accomplished by having the CPU hardware compare address generated in user mode with the registers, This scheme prevents a user program from modifying the code or data structures of either the operating system or other users.

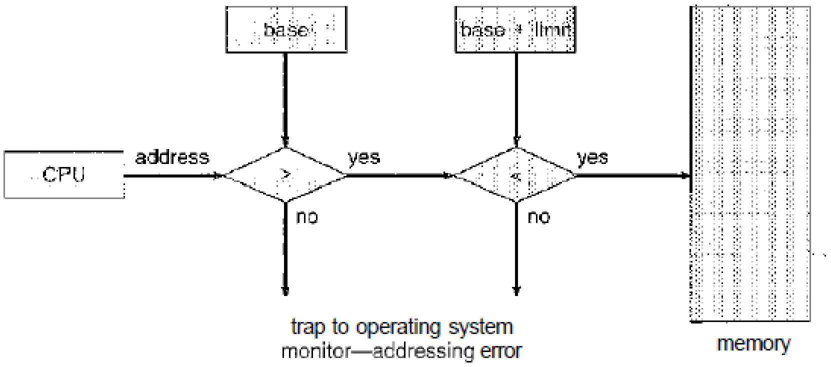


Figure 8.2 Hardware address protection with base and limit registers.

Address Binding

Usually, a program resides on a disk as a binary executable file. To be executed, the program must be brought into memory and placed within a process Depending on the memory management in use, **the process may be moved between disk and memory during its execution** The processes on the disk that are waiting to be brought into memory for execution form the input queue.

The normal procedure is to select one of the processes in the input queue and to load that process into memory. As the process is executed, it accesses instructions and data from memory. Eventually, the process terminates, and its memory space is declared available.

Most systems allow a user process to reside in any part of the physical memory. Thus, although the address space of the computer starts at 00000, the first address of the user process need not be 00000.

Compile time

If you know at compile time where the process will reside in memory, then absolute code can be generated. For example, if you know that a user process will reside starting at location R, then the generated compiler code will start at that location and extend up from there. If, at some later time, the starting location changes, then it will be necessary to recompile this code.

Load Time

If it is **not known** at compile time where the process will reside in memory, then the compiler must generate **relocatable code**. In this case, final binding is delayed until load time. If the starting address changes, we need only reload the user code to incorporate this changed value.

Execution time If the process **can be moved during its execution from one memory segment to another**, then binding must be delayed until run time. Special hardware must be available for this scheme to work.

Logical Versus Physical Address Space

An address generated by the CPU is commonly referred to as a **logical address**, whereas an address seen by the **memory unit**—that is, the one loaded into the **memory-address register** of the memory—is commonly referred to as a **physical address**.

Note: The compile-time and load-time address-binding methods generate identical logical and physical addresses.

However, the **execution-time** address binding scheme results in **differing logical and physical addresses** In this case, we usually refer to the logical address as a **virtual address**.

Note: The set of all logical addresses generated by a program is a logical address space; the set of all physical addresses corresponding to these logical addresses is a physical address space

Thus, in the execution-time address-binding scheme, the logical and physical address spaces differ. The run-time mapping from virtual to physical addresses is done by a hardware device called the **memory-management unit (MMU)**.

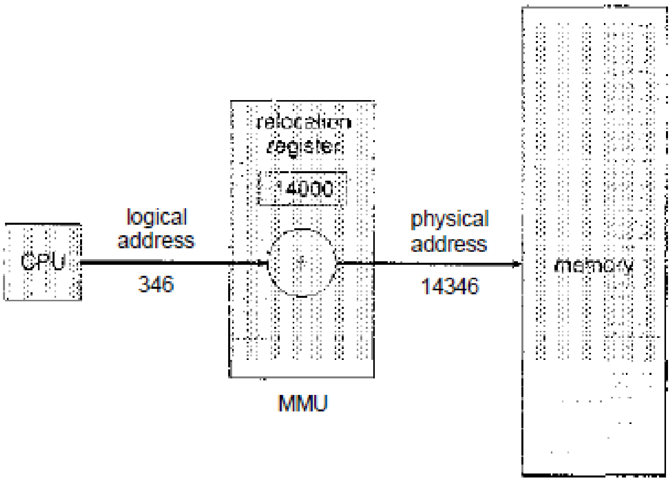


Figure 8.4 Dynamic relocation using a relocation register.

The base register is now called a **relocation register**. The value in the relocation register is added to every address generated by a user process at the time it is sent to memory.
Note: We now have two different types of addresses: logical addresses (in the range 0 to max) and physical addresses (in the range R + 0 to R + max for a **base value R**).

Dynamic Loading

The entire program and all data of a process must be in physical memory for the process to execute. The size of a process is thus limited to the size of physical memory. To obtain better memory-space utilization, we can use dynamic loading. With dynamic loading, a routine is not loaded until it is called. All routines are kept on disk in a relocatable load format. The main program is loaded into memory and is executed. When a routine needs to call another routine, the calling routine first checks to see whether the other routine has been loaded. If not, the **relocatable linking loader** is called to load the desired routine into memory and to update the program's address tables to reflect this change

The advantage of dynamic loading is that an unused routine is never loaded. **Dynamic loading does not require special support from the operating system**. It is the responsibility of the users to design their programs to take advantage of such a method.

Dynamic Linking and Shared Libraries:

Some operating systems support only **static linking**, in which system language libraries are treated like any other object module and are combined by the loader into the binary program image. **Linking** is postponed until execution time. This feature is usually used with system libraries, such as language subroutine libraries. With dynamic linking, a **stub** is included in the image for each library routine reference. The stub is a small piece of code that indicates how to locate the appropriate memory-resident library routine or how to load the library if the routine is not already present

Note: Unlike dynamic loading, dynamic linking generally requires help from the operating system.

Swapping:

A process must be in memory to be executed. A process, however, can be swapped temporarily out of memory to a **backing store** and then brought back into memory for continued execution. For example, assume a multiprogramming environment with a round-robin CPU-scheduling algorithm. When a quantum expires, the memory manager will start to swap out the process that just finished and to swap another process into the memory space that has been freed. In the meantime, the CPU scheduler will allocate a time slice to some other process in memory. When each process finishes its quantum, it will be swapped with another process. Ideally, the memory manager can swap processes fast enough that some processes will be in memory, ready to execute, when the CPU scheduler wants to reschedule the CPU. In addition, the quantum must be large enough to allow reasonable amounts of computing to be done between swaps. A variant of this swapping policy is used for priority-based scheduling algorithms. If a higher-priority process arrives and wants service, the memory manager can swap out the lower-priority process and then load and execute the higher-priority process. When the higher-priority process finishes, the lower-priority process can be swapped back in and continued. This variant of swapping is sometimes called **roll out, roll in**.

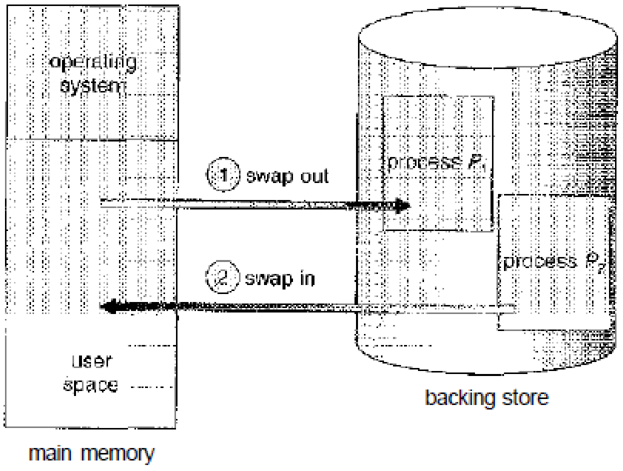


Figure 8.5 Swapping of two processes using a disk as a backing store.

Normally, a process that is swapped out will be swapped back into the same memory space it occupied previously. This restriction is dictated by the method of address binding. If binding is done at assembly or load time, then the process cannot be easily moved to a different location. If execution time binding is being used, however, then a process can be swapped into a different memory space, because the physical addresses are computed during execution time.

Memory Allocation

One of the simplest methods for allocating memory is to divide memory into several fixed-sized Partitions. Each partition may contain exactly one process. Thus, the degree of multiprogramming is bound by the number of partitions. In this multiple-partition method, when a partition is free, a process is selected from the input queue and is loaded into the free partition. When the process terminates, the partition becomes available for another process.

In the **fixed-partition scheme**, the operating system keeps a table indicating which parts of memory are available and which are occupied. Initially, all memory is available for user processes and is considered one large block of available memory, **a hole**. When a process arrives and needs memory, we search for a hole large enough for this process. If we find one, we allocate only as much memory as is needed, keeping the rest available to satisfy future requests.

In general, at any given time we have a set of holes of various sizes scattered throughout memory. When a process arrives and needs memory, the system searches the set for a hole that is large enough for this process. If the hole is too large, it is split into two parts. One part is allocated to the arriving process; the other is returned to the set of holes. When a process terminates, it releases its block of memory, which is then placed back in the set of holes. If the new hole is adjacent to other holes, these adjacent holes are merged to form one larger hole.

This procedure is a particular instance of the general **dynamic storage allocation** problem, which concerns how to satisfy a request of size n from a list of free holes. There are many solutions to this problem. The **first-fit**, **best-fit**, and **worst-fit** strategies are the ones most commonly used to select a free hole from the set of available holes.

First fit Allocate the first hole that is big enough. Searching can start either at the beginning of the set of holes or where the previous first-fit search ended (**Next fit**). We can stop searching as soon as we find a free hole that is large enough.
Best fit Allocate the smallest hole that is big enough. **We must search the entire list, unless the list is ordered by size.** This strategy produces the smallest leftover hole.
Worst fit Allocate the largest hole. Again, we must search the entire list, unless it is sorted by size. This strategy produces the largest leftover hole, **which may be more useful than the smaller leftover hole from a best-fit approach**.

Note:

1. Both “first fit” and “best fit” are **better than** worst fit in terms of “decreasing time” and storage utilization.
2. Neither first-fit nor best-fit is clearly better than the other in terms of storage utilization.
3. first fit is generally faster.
4. Both the first-fit and best-fit strategies for memory allocation suffer from **external fragmentation**.

Fragmentation

As processes are loaded and removed from memory, the free memory space is broken into little pieces. External fragmentation exists when there is enough total memory space to satisfy a request, but the available spaces are not contiguous.

One solution to the problem of external fragmentation is **compaction**. The goal is to shuffle the memory contents so as to place all free memory together in one large block. Compaction is not always possible, however. If relocation is static and is done at assembly or load time, compaction cannot be done; compaction is possible only if relocation is dynamic and is done at execution time. Another **possible solution to the external-fragmentation problem is to permit the logical address space of the processes to be non-contiguous**.

Internal fragmentation is memory that is internal to a partition but is not being used.

PAGING

Paging is a memory-management scheme that permits the physical address space of a process to be noncontiguous. When some code fragments or data residing in main memory need to be swapped out, space must be found on the backing store. **The backing store also has the fragmentation problems**.

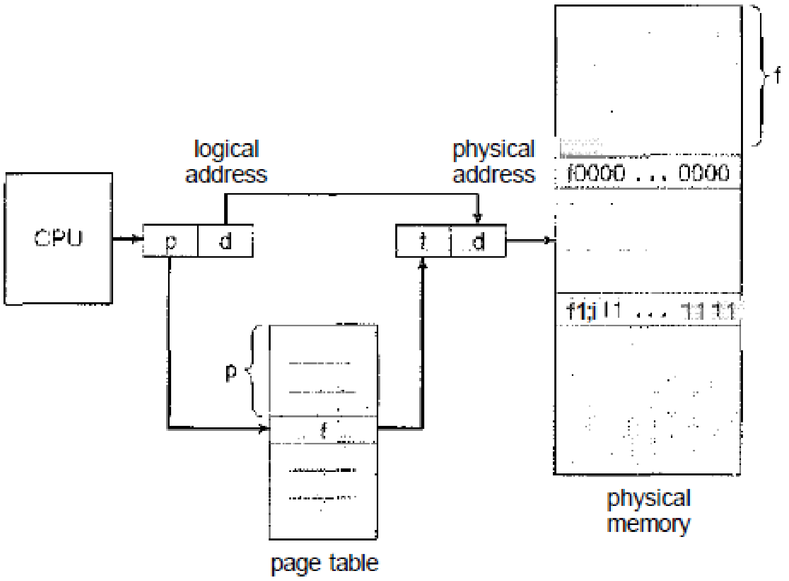


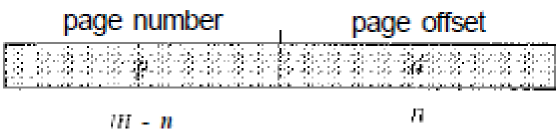
Figure 8.7 Paging hardware.

Basic Method

The basic method for implementing paging involves breaking physical memory into fixed-sized blocks called **frames** and breaking logical memory into blocks of the same size called **pages**. When a process is to be executed, its pages are loaded into any available memory frames from the backing store. The backing store is divided into fixed-sized blocks that are of the same size as the memory 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:



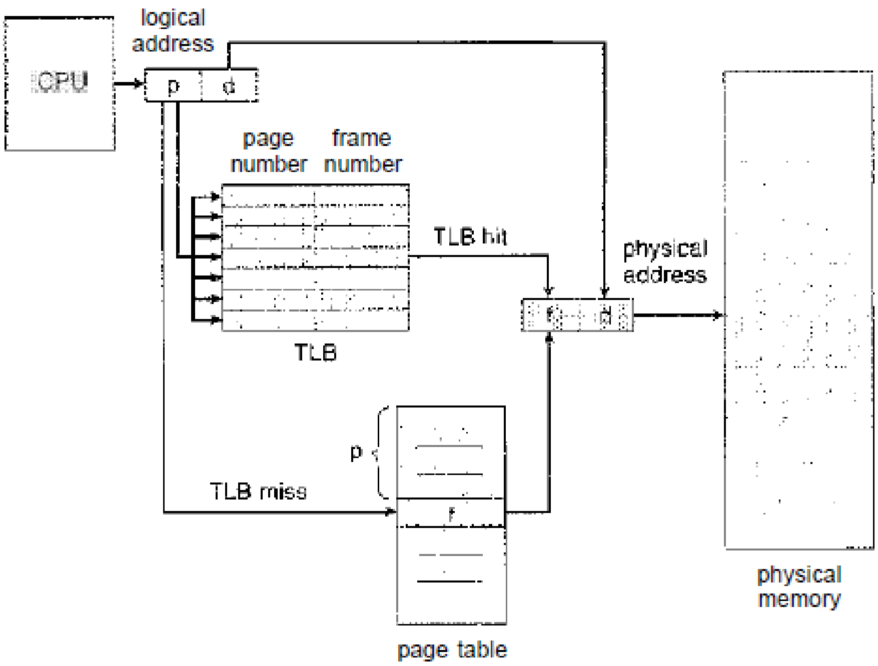


Figure 8.11 Paging hardware with TLB.

The TLB contains only a few of the page-table entries. When a logical address is generated by the CPU, its page number is presented to the TLB. If the page number is found, its frame number is immediately available and is used to access memory. The whole task may take less than 10 percent longer than it would if an unmapped memory reference were used.

If the page number is not in the TLB (known as a TLB miss), a memory reference to the page table must be made. When the frame number is obtained, we can use it to access memory (Figure 8.11). In addition, we add the page number and frame number to the TLB, so that they will be found quickly on the next reference. If the TLB is already full of entries, the operating system must select one for replacement. Replacement policies range from least recently used (LRU) to random.

Some TLBs allow entries to be wired down, meaning that they cannot be removed from the TLB. Typically, TLB entries for kernel code are wired down.

TLB - Address Space Identifiers:

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.
It 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 the TLB must be flushed/erased to ensure that the next executing process does not use the wrong translation information.

TLB E.M.A.T = TLB_Hit_Ratio*(TLB A.T + M/M A.T) + (1- TLB_Hit_Ratio)*(TLB A.T + 2*M/M A.T)

TLB, Cache and Page fault relationship:- (Single level Paging)

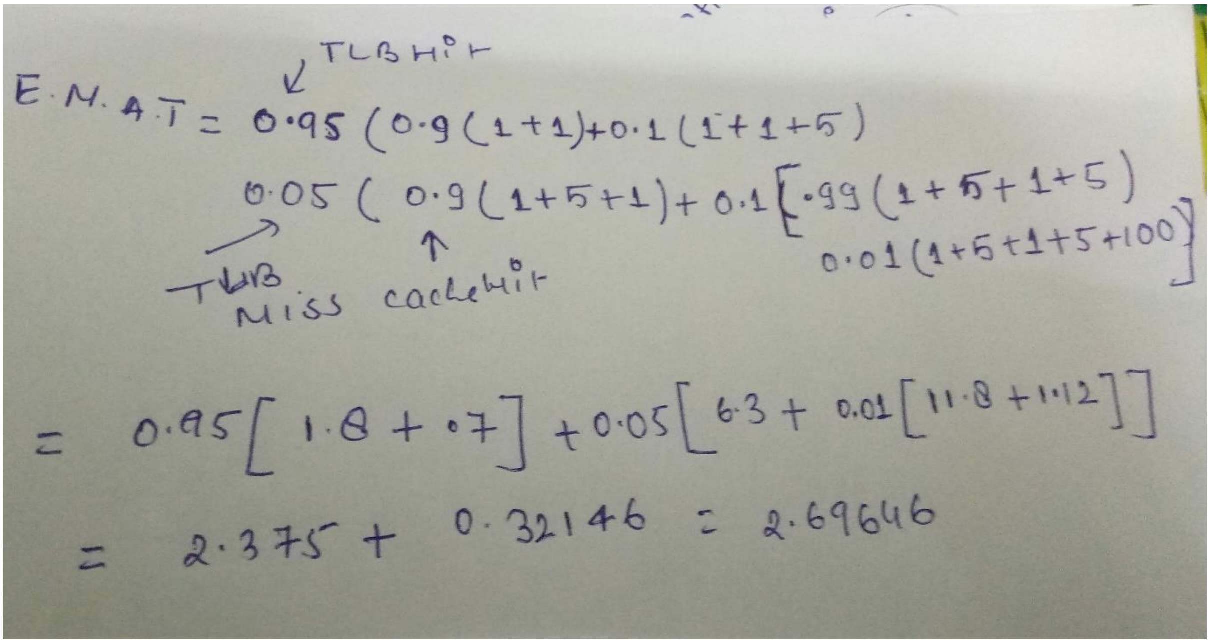
E.M.A.T = TLB_Hit_Rate(Cache_Hit_Rate(TLB A.T + Cache A.T) + Cache_Miss_Rate(TLB A.T + Cache A.T + M/M A.T)) + TLB_Miss_Rate(Cache_Hit_Rate(TLB A.T + Page Table A.T + Cache A.T) + Cache_Miss_Rate(No_Page_fault(TLB A.T + Page Table A.T + Cache A.T + M/M A.T) + PageFaultRate(TLB A.T + Page Table A.T + Cache A.T + M/M A.T + Page Fault Time))

Example:

Assume cache is physically addressed cache, consider the following information:

1. TLB HIT Rate 95%, Access Time is 1 cycle
2. Cache Hit Ratio 90%, Access Time is 1 cycle
3. When TLB and Cache both get missed page fault is 1%
4. TLB and cache access are sequential
5. Main m/m access is 5 cycles
6. PFST is 100 cycles
7. Page table is kept in main memory

What is E.M.A.T?



TLB Hit

$$E.M.A.T = 0.95(0.9(1+1)+0.1(1+1+5)$$

$$0.05(0.9(1+5+1)+0.1[.99(1+5+1+5)$$

$$0.01(1+5+1+5+100)]$$

TLB Miss cache hit

$$= 0.95[1.8+.7]+0.05[6.3+0.01[11.8+1.12]]$$

$$= 2.375 + 0.32146 = 2.69646$$

Protection

Memory protection in a paged environment is accomplished by protection bits associated with each frame. Normally, these bits are kept in the page table. One bit can define a page to be read-write or read-only. Every reference to memory goes through the page table to find the correct frame number. At the same time that the physical address is being computed, the protection bits can be checked to verify that no writes are being made to a read-only page. An attempt to write to a read-only page causes a hardware trap to the operating system (or memory-protection violation).

Shared Pages

An advantage of paging is the possibility of **sharing common code**. If the code is **reentrant code** (or **pure code**), it can be shared. **Reentrant code is non-self-modifying code**; **it never changes during execution**. Thus, two or more processes can execute the same code at the same time.

Structure of the Page Table:

Hierarchical Paging: Most modern computer systems support a large logical address space (2^32 to 2^64). In such an environment, the page table itself becomes excessively large. For example, consider a system with a 32-bit logical address space. If the page size in such a system is 4 KB (2^12), then a page table may consist of up to 1 million entries (2^32/2^12). Assuming that each entry consists of 4 bytes, each process may need up to **4 MB of physical address space** for the page table alone.

One way is to use a two-level paging algorithm, in which the page table itself is also paged. Our example of a 32-bit machine with a page size of 4 KB. A logical address is divided into a page number consisting of 20 bits and a page offset consisting of 12 bits. Because we page the page table, the page number is further divided into a 10-bit page number and a 10-bit page offset. Thus, a logical address is as follows:

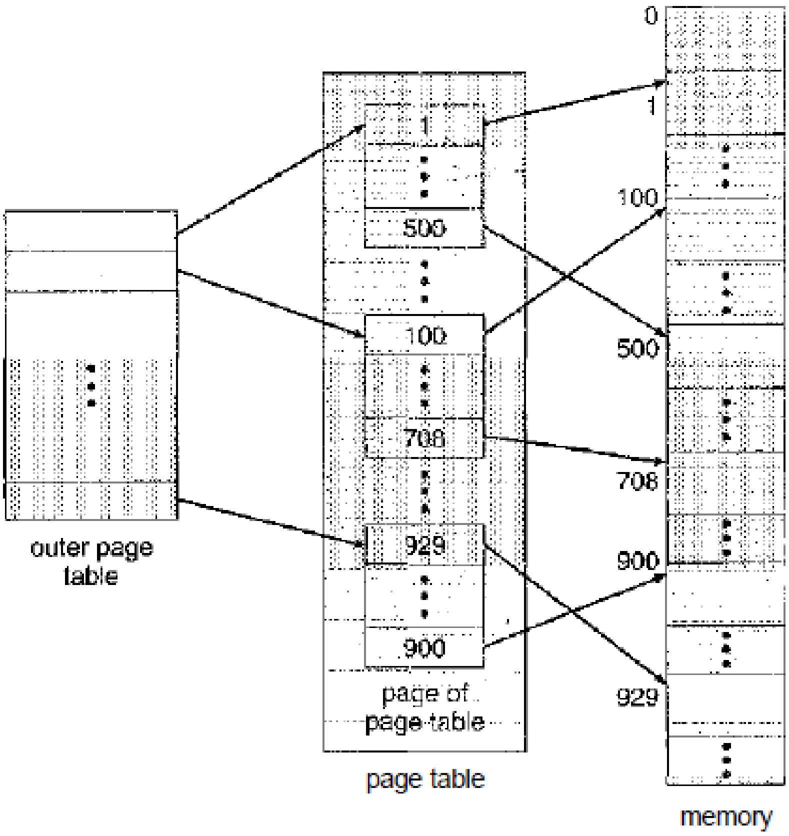
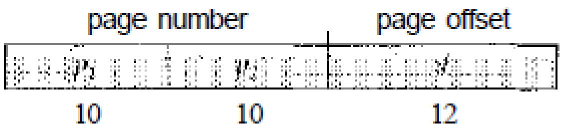


Figure 8.14 A two-level page-table scheme.



Where p1 is an index into the outer page table and **p2 is the displacement within the page of the outer page table**. The address-translation method for this architecture is shown in Figure 8.15. Because address translation works from the outer page table inward, this scheme is also known as a **forward-mapped page table**.

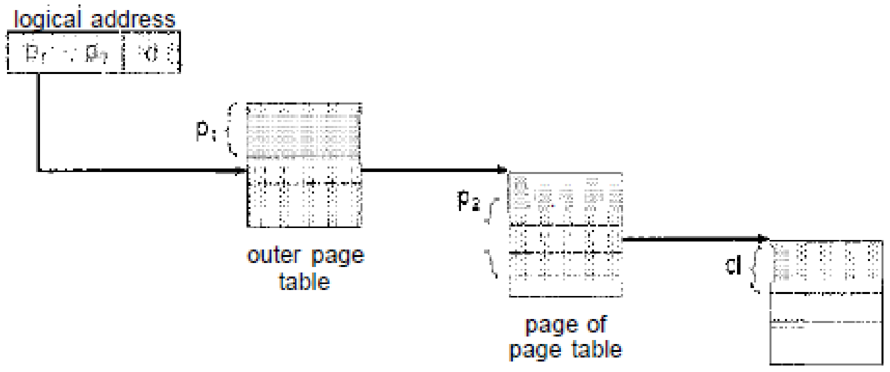
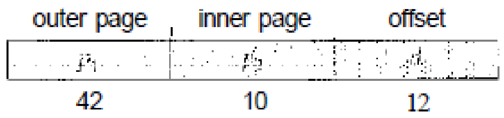


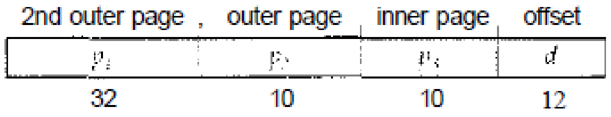
Figure 8.15 Address translation for a two-level 32-bit paging architecture.

For a system with a 64-bit logical-address space, a two-level paging scheme is no longer appropriate. To illustrate this point, let us suppose that the page size in such a system is 4 KB (2^{12}). In this case, the page table consists of up to 2^{52} entries. If we use a two-level paging scheme, then the inner page tables can conveniently be one page long, or contain 2^{10} 4-byte entries. The addresses look like this:



The outer page table consists of 2^{42} entries, or 2^{44} bytes. The obvious way to avoid such a large table is to divide the outer page table into smaller pieces. This approach is also used on some 32-bit processors for added flexibility and efficiency.

We can divide the outer page table in various ways. We can page the outer page table, giving us a three-level paging scheme. Suppose that the outer page table is made up of standard-size pages (2^{10} entries, or 2^{12} bytes); a 64-bit address space is still daunting:



The outer page table is still 2^{34} bytes in size.

Inverted Page Tables

Usually, **each process has an associated page table**. The page table has one entry for each page that the process is using. An inverted page table has one entry for each **real page (or frame) of memory** each entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page. Thus, only one page table is in the system, and it has only one entry for each page of physical memory.

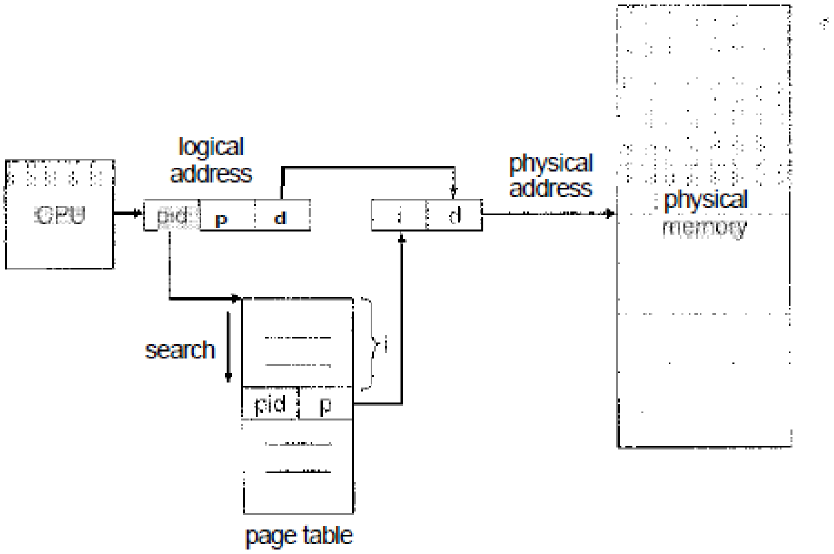


Figure 8.17 Inverted page table.

Inverted page tables often require that an address-space identifier be stored in each entry of the page table, since the table usually contains several different address spaces mapping physical memory. Storing the address-space identifier ensures that a logical page for a particular process is mapped to the corresponding physical page frame.

<process-id, page-number, offset>

Each inverted page-table entry is a pair <process-id, page-number> where the process-id assumes the role of the address-space identifier. When a memory reference occurs, part of the virtual address, consisting of <process-id, page number>, is presented to the memory subsystem. The inverted page table is then searched for a match. If a match is found say, at **entry i** then the physical address **<i, offset>** is generated. If no match is found, then an illegal address access has been attempted. Although this scheme decreases the amount of memory needed to store each page table, it increases the amount of time needed to search the table when a page reference occurs. Because the inverted page table is sorted by physical address, but lookups occur on virtual addresses, the whole table might need to be searched for a match. This search would take far too long. **To solve this problem, we use a hash table.**

Note: Systems that use inverted page tables have difficulty implementing shared memory.

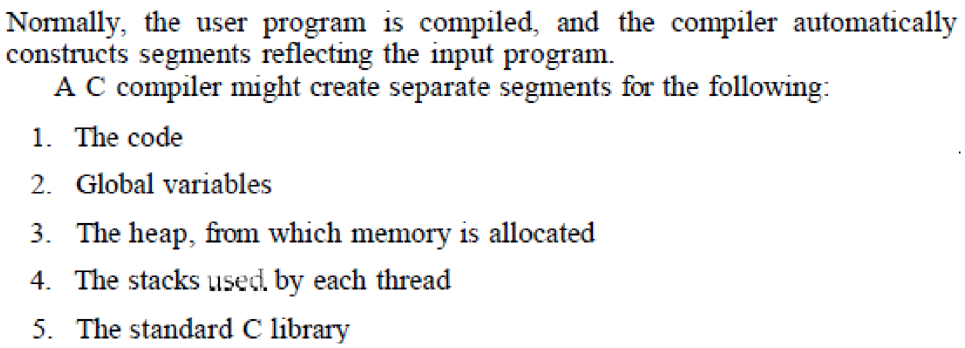
Segmentation

Basic Method

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 name and a length**. The addresses specify both the **segment name and the offset within the segment**. The user therefore specifies each address by two quantities: a segment name and an offset.

For simplicity of implementation, segments are numbered and are referred to by a segment number, rather than by a segment name. Thus, a logical address consists of a two tuple:

<segment-number, offset>



Normally, the user program is compiled, and the compiler automatically constructs segments reflecting the input program.

A C compiler might create separate segments for the following:

1. The code
2. Global variables
3. The heap, from which memory is allocated
4. The stacks used by each thread
5. The standard C library

Mapping is effected by a segment table. **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.

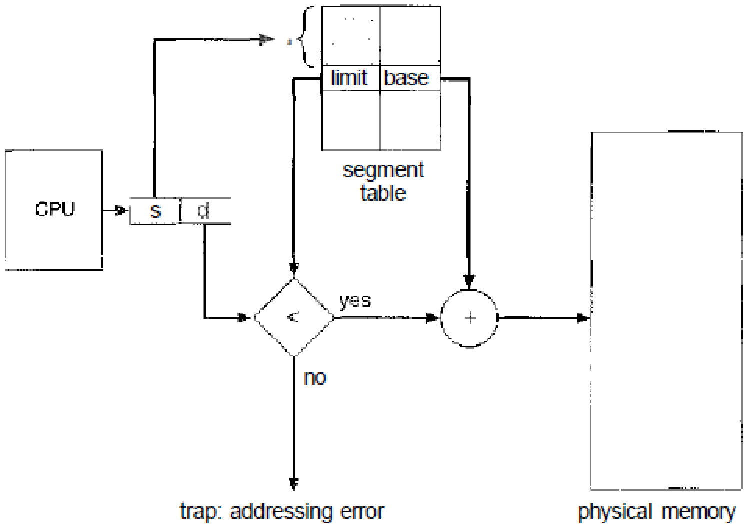


Figure 8.19 Segmentation hardware.

A logical address consists of two parts: a **segment number s**, and an **offset into that segment d**. The segment number is used as an index to the segment table. The offset d of the logical address must be between 0 and the segment limit. If it is not, we trap to the operating system. When an offset is legal, it is added to the segment base to produce the address in physical memory of the desired byte.

Note: The segment table has a separate entry for each segment, giving the beginning address of the segment in physical memory (or base) and the length of that segment (or limit).

Virtual Memory

Demand Paging

A demand-paging system is similar to a paging system with swapping where processes reside in secondary memory. When we want to execute a process, we swap it into memory. Rather than swapping the entire process into memory, however, we use a **lazy swapper**. A lazy swapper never swaps a page into memory unless that page will be needed. We are now viewing a process as a sequence of pages.

With this scheme, we need some form of hardware support to distinguish between the pages that are in memory and the pages that are on the disk.

Page Fault:

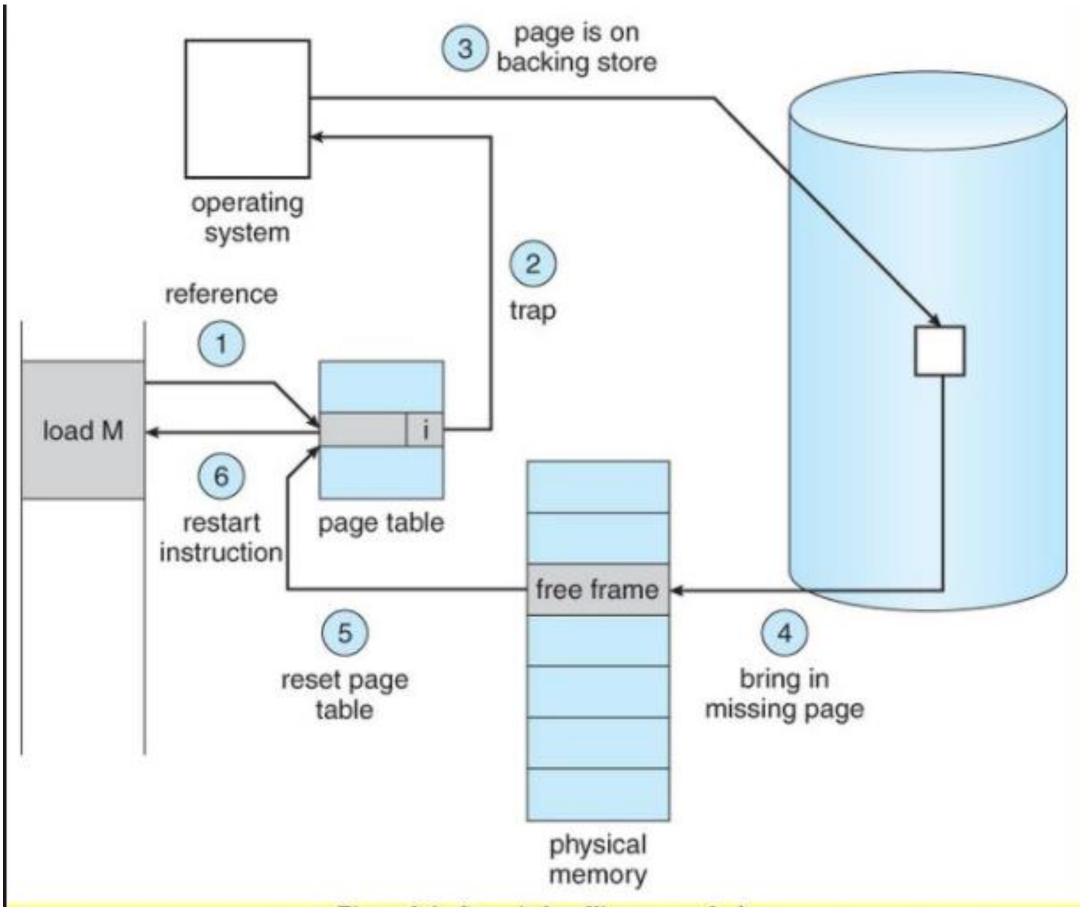


Figure 9.6 - Steps in handling a page fault

Performance of Demand Paging

Let p be the probability of a page fault (0<=p 5 <=1), ma is the memory access time.

Effective Access Time: (1-p)*ma + p*Page Fault Time

Basic Page Replacement:

We modify the page-fault service routine to include page replacement:

1. Find the location of the desired page on the disk.
2. Find a free frame:
   a. If there is a free frame, use it.
   b. If there is no free frame, use a page-replacement algorithm to select a victim frame.
   c. Write the victim frame to the disk; change the page and frame tables accordingly.
3. Read the desired page into the newly freed frame; change the page and frame tables.
4. Restart the user process.

Notice that, if no frames are free, two page transfers (one out and one in) are required. This situation effectively **doubles** the page-fault service time. We can reduce this overhead by using a **modify bit** (**or dirty bit**).
The string of memory references is called a **reference string**. We need to **consider only the page number, rather than the entire address**.

For example, if we trace a particular process, we might record the following address sequence:
0100, 0432, 0101, 0612, 0102, 0103, 0104, 0101, 0611, 0102, 0103, 0104, 0101, 0610, 0102, 0103, 0104, 0101, 0609, 0102, 0105

At 100 bytes per page, this sequence is reduced to the following reference string:

1, 4, 1, 6, 1, 6, 1, 6, 1

Note: the number of frames available increases, the number of page faults decreases.

FIFO Page Replacement:

A FIFO replacement algorithm associates with each page the time when that page was brought into memory. When a page must be replaced, the oldest page is chosen. Notice that it is not strictly necessary to record the time when a page is brought in. We can create a FIFO queue to hold all pages in memory. We replace the page at the head of the queue. When a page is brought into memory, we insert it at the tail of the queue. **Belady's anomaly**: For some page-replacement algorithms, the page-fault rate may increase as the number of allocated frames increases.

Optimal Page Replacement

An optimal page-replacement algorithm has the **lowest page-fault rate of all algorithms** and will never suffer from Belady's anomaly.

"Replace the page that will not be used for the longest period of time."

Unfortunately, the optimal page-replacement algorithm is difficult to implement, because it requires future knowledge of the reference string.

LRU Page Replacement

"Replace the page that has not been used for the longest period of time"

We can think of this strategy as the optimal page-replacement algorithm looking backward in time, rather than forward.

Note:

1. **LRU and Optimal Page replacement algorithms, give the same number of page fault on the reference string S and its reverse S^r**.
2. **An LRU page-replacement algorithm may require substantial hardware assistance.**
3. **LRU replacement does not suffer from Belady's anomaly**

LRU and OPT both are called **stack algorithms** A stack algorithm is an algorithm for which it can be shown that the set of pages in memory for n frames is always a subset of the set of pages that would be in memory with n + 1 frames. For LRU, the set of pages in memory would be the n most recently referenced pages.

Counting-Based Page Replacement

1. **The least frequently used (LFU) page-replacement** algorithm requires that the page with the smallest count be replaced. A problem arises, however, when a page is used heavily during the initial phase of a process but then is never used again. Since it was used heavily, it has a large count and remains in memory even though it is no longer needed.
2. **The most frequently used (MFU) page-replacement algorithm** is based on the argument that the page with the smallest count was probably just brought in and has yet to be used.

The implementation of these algorithms is expensive, and they do not approximate OPT replacement well.

Thrashing

If the process does not have the number of frames it needs to support pages in **active use**, it will quickly page-fault. At this point, it must replace some page. However, since all its pages are in active use, it must replace a page that will be needed again right away. Consequently, it quickly faults again, and again, and again, replacing pages that it must bring back in immediately.

This high paging activity is called “thrashing”. A process is thrashing if it is spending more time paging than executing.

Cause of Thrashing

If CPU utilization is too low, we increase the degree of multiprogramming by introducing a new process to the system. A global page-replacement algorithm is used; it replaces pages without regard to the process to which they belong. Now suppose that a process enters a new phase in its execution and needs more frames. It starts faulting and taking frames away from other processes. These processes need those pages, however, and so they also fault, taking frames from other processes. These faulting processes must use the paging device to swap pages in and out. As they queue up for the paging device, the ready queue empties. As processes wait for the paging device, CPU utilization decreases.

The CPU scheduler sees the decreasing CPU utilization and increases the degree of multiprogramming as a result. The new process tries to get started by taking frames from running processes, causing more page faults and a longer queue for the paging device. As a result, CPU utilization drops even further, and the CPU scheduler tries to increase the degree of multiprogramming even more.

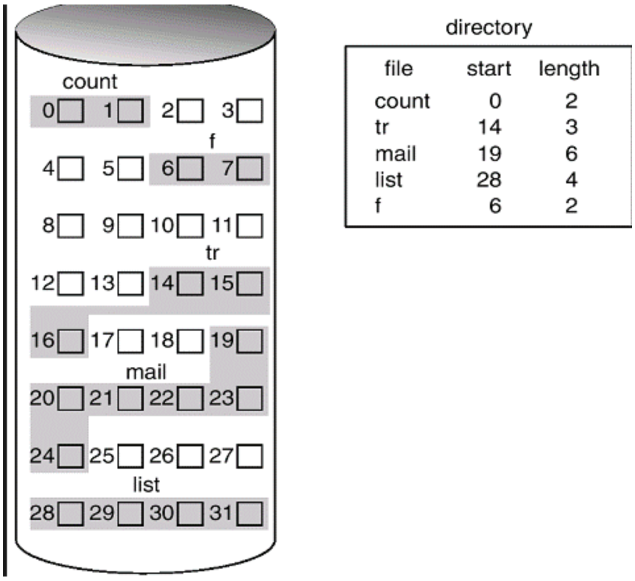
Note: We can limit the effects of thrashing by using a **local replacement algorithm** (or **priority replacement algorithm**). With local replacement, if one process starts thrashing, it cannot steal frames from another process, but not a complete solution it is.
To prevent thrashing, we must provide a process with as many frames as it needs.

Buddy System:

Assume the size of a memory segment is initially 256 KB and the kernel requests 21 KB of memory. It will divide in two buddies of size 128KB and 128KB, then further It will divide one buddy into two parts of 64KB, then further into two parts of 32KB and then one part will be assigned to21KB process. If required two buddies can be merged to make a bigger size.

Allocation Methods

1. **Contiguous Allocation:** Contiguous allocation requires that each file occupy a set of contiguous blocks on the disk. Disk addresses define a linear ordering on the disk. With this ordering, assuming that only one job is accessing the disk, accessing block b+1 after block b normally requires no head movement. When head movement is needed (from the last sector of one cylinder to the first sector of the next cylinder), the head need only move from one track to the next. Thus, the number of disk seeks required for accessing contiguously allocated files is minimal. Contiguous allocation of a file is **defined by the disk address and length** (in block units) of the first block.



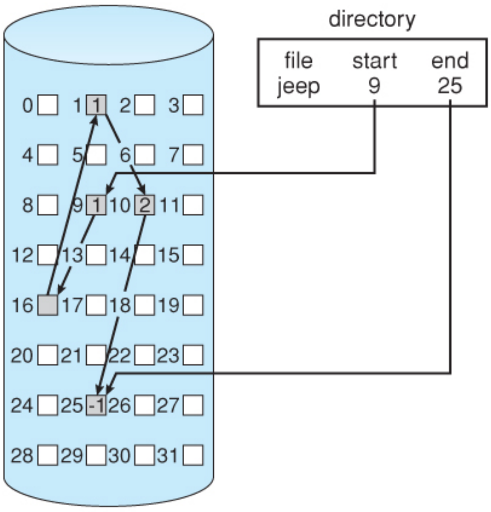
directory

file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

Note: It suffers from the problem of external fragmentation.
Compaction scheme effectively **compacts** all free space into one contiguous space, solving the fragmentation problem.

Linked Allocation

It solves all problems of contiguous allocation. With linked allocation, each file is a linked list of disk blocks; the disk blocks may be scattered anywhere on the disk. The directory contains a pointer to the first and last blocks of the file. Each block contains a pointer to the next block. These pointers are not made available to the user. Thus, if each block is 512 bytes in size, and a disk address (the pointer) requires 4 bytes, then the user **sees blocks of 508 bytes.** To create a new file, we simply create a new entry in the directory. With linked allocation, each directory entry has a pointer to the first disk block of the file. A file can continue to grow as long as free blocks are available. it is never necessary to compact disk space.



The major problem is that it can be used effectively only for **sequential-access** files, and the space required for the pointers, if a pointer requires 4 bytes out of a 512-byte block, then 0.78 percent of the disk is being used for pointers. Another problem of linked allocation is reliability, what would happen if a pointer were lost or damaged.

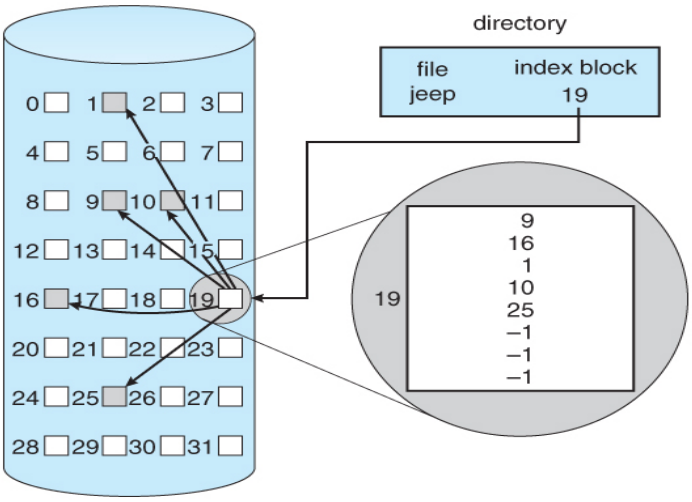
Note: The usual solution to this problem is to collect blocks into multiples, **called clusters**, and to allocate clusters rather than blocks.

An important variation on linked allocation is the use of a **file-allocation table (FAT)**.

Indexed Allocation

Indexed allocation solves this problem by bringing all the pointers together into one location: **the index block**. Each file has its own index block, which is an array of disk-block addresses. The ith entry in the index block points to the ith block of the file. The directory contains the address of the index block, to find and read the ith block, we use the pointer in the ith index-block entry. When the file is created, all pointers in the index block are set to nil.

Indexed allocation supports **direct access**, **without suffering from external fragmentation**, because any free block on the disk can satisfy a request for more space. **Indexed allocation does suffer from wasted space**. **The pointer overhead of the index block is generally greater than the pointer overhead of linked allocation**. Consider a common case in which we have a file of only one or two blocks. With linked allocation, we lose the space of only one pointer per block. With indexed allocation, an entire index block must be allocated, even if only one or two pointers will be non-nil.



Free-Space Management

We need to reuse the space from deleted files for new files, if possible. To keep track of free disk space, the system maintains a **free-space list**. The free-space list records all free disk blocks—those not allocated to some file or directory. When a file is deleted, its disk space is added to the free-space list.

1. Bit Vector

The free-space list is implemented as a bit map or bit vector. Each block is represented by 1 bit. If the block is free, the bit is 1; if the block is allocated, the bit is 0.

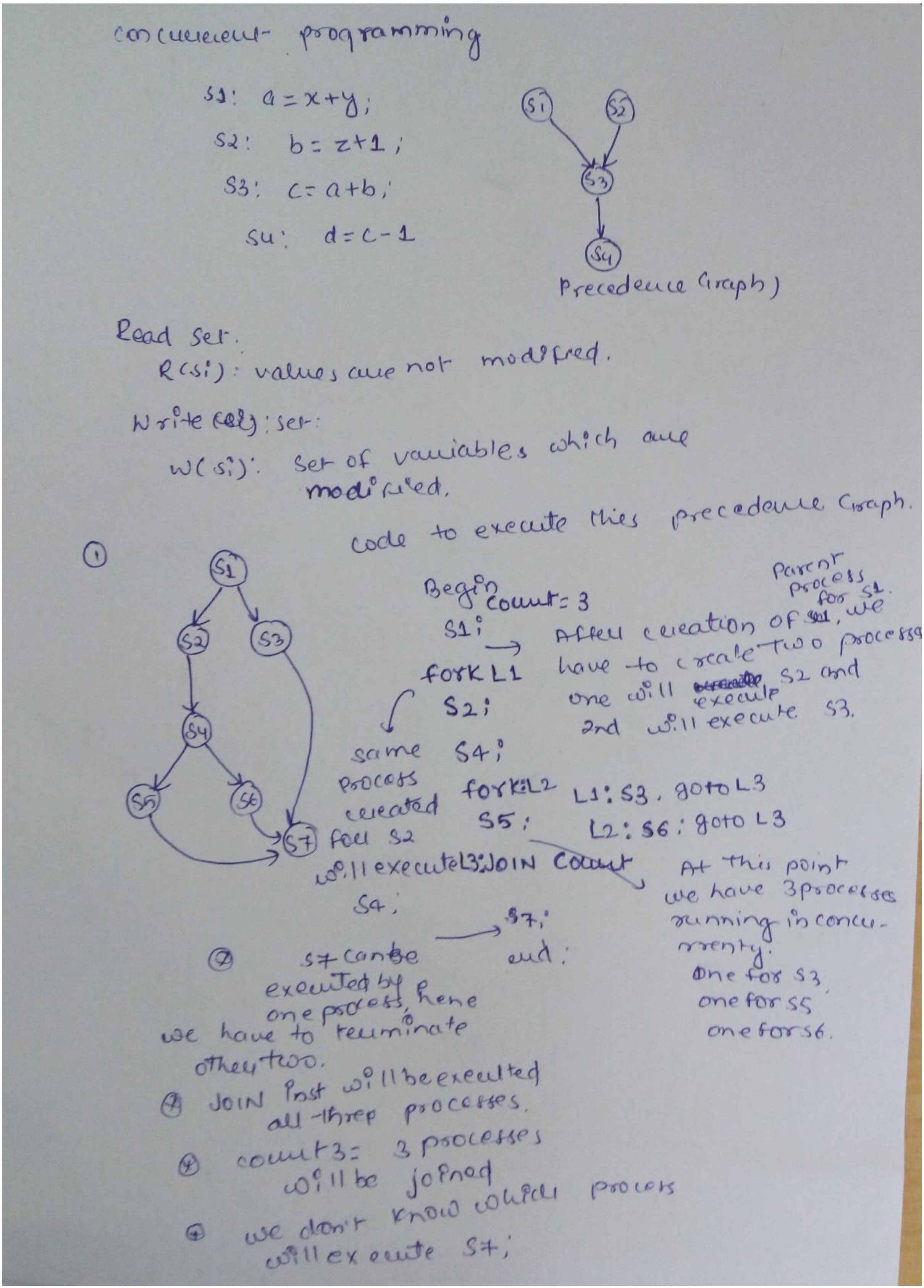
001111001111111000

The main advantage of this approach is its relative simplicity and its efficiency in finding the first free block or n consecutive free blocks on the disk.

2. Linked List

Another approach to free-space management is to link together all the free disk blocks. This first block contains a pointer to the next free disk block, and so on.

Concurrent Program Execution


S1
S2
S3
S4
S5
S6
S7

concurrent programming

```
S1:  a = x+y;
S2:  b = z+1;
S3:  c = a+b;
S4:  d = c-1
```

Precedence Graph)

Read set.
R(Si): values are not modified.

Write ~~(W)~~: set:
W(Si): set of variables which are modified.

① code to execute this precedence graph.

```
Begin count = 3
  S1;
  fork L1
  S2;
  S4;
  fork L2
  S5;
L3: JOIN count
  S7;
end;
L1: S3. goto L3
L2: S6: goto L3
```

S1; → After creation of ~~S1~~, we have to create two process one will ~~execute~~ execute S2 and 2nd will execute S3.

Parent process for S1.

same process created for S2 will execute S4;

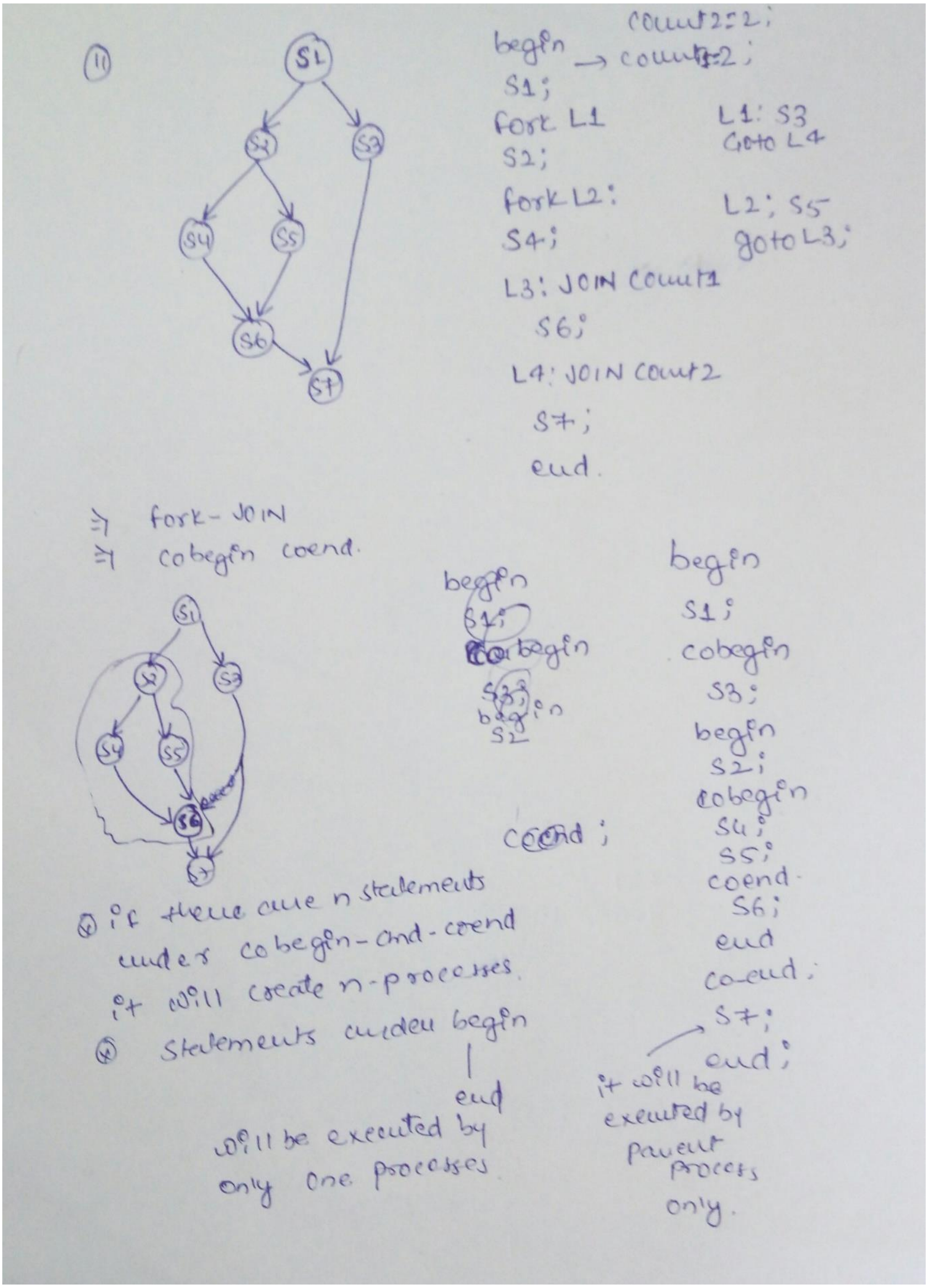
At this point we have 3 processes running in concurrently.
One for S3,
one for S5
one for S6.

② S7 can be executed by one process, hence we have to terminate other two.

③ JOIN first will be executed all three processes.

④ count 3 = 3 processes will be joined

⑤ we don't know which process will execute S7;


S1
S2
S3
S4
S5
S6
S7
S1
S2
S3
S4
S5
S6
S7

(11)

```
begin → count1=2;
        count2=2;
S1;
fork L1                L1: S3
S2;                    Goto L4
fork L2:               L2; S5
S4;                    goto L3;
L3: JOIN count1
  S6;
L4: JOIN count2
  S7;
  end.
```

⇒ fork-JOIN

⇒ cobegin coend.

```
begin
S1;
cobegin
S3;
begin
S2

coend ;
```

```
begin
S1;
cobegin
 S3;
 begin
  S2;
  cobegin
   S4;
   S5;
  coend.
  S6;
 end
co-end;
S7;
end;
```

it will be executed by parent process only.

① if there are n statements under cobegin-and-coend it will create n-processes.

② Statements under begin | end will be executed by only one processes

Live Lock:

A livelock is similar to a deadlock, except that the states of the processes involved in the livelock **constantly change with regard to one another**, none progressing. Livelock is a special case of resource starvation; the general definition only states that a specific process is not progressing.

A real-world example of livelock occurs when two people meet in a narrow corridor, and each tries to be polite by moving aside to let the other pass, but they end up swaying from side to side without making any progress because they both repeatedly move the same way at the same time.

File Structure:

Single-Level Directory: The simplest directory structure is the single-level directory. All files are contained in the same directory. Since all files are in the same directory, they must have unique names.

Two-Level Directory:

The standard solution is to create a separate directory for each user. Each user has his own user file directory.

It still has disadvantages. This structure effectively isolates one user from another. Isolation is an advantage when the users are completely independent but is a disadvantage when the users want to cooperate on some task and to access one another's files. Some systems simply do not allow local user files to be accessed by other users.

Tree-Structured Directories:

Multilevel directory. The tree has a root directory, and every file in the system has a unique path name. One bit in each directory entry defines the entry as a file (0) or as a subdirectory (1). Special system calls are used to create and delete directories.

An **absolute path** name begins at the root and follows a path down to the specified file, giving the directory names on the path. A **relative path** name defines a path from the current directory.

Acyclic-Graph Directories:

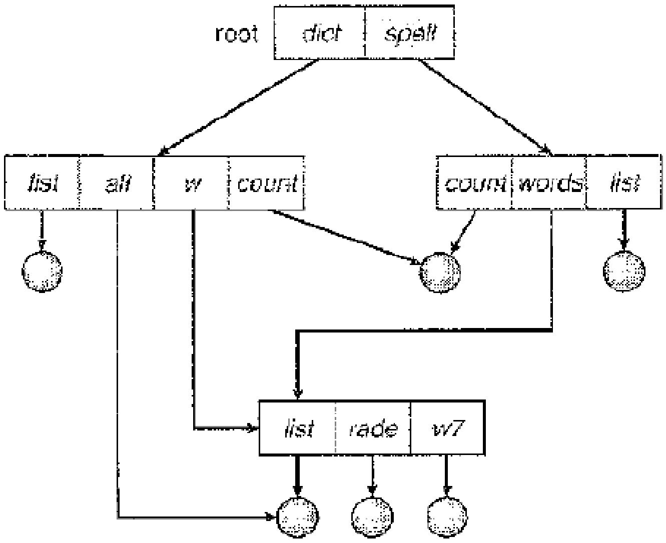


Figure 10.10 Acyclic-graph directory structure.

A tree structure prohibits the sharing of files or directories.

An **acyclic graph** —that is, a graph with no cycles—allows directories to share subdirectories and files. Whenever anyone deletes it, but this action may leave **dangling pointers** to the now non-existent file. With a shared file, only one actual file exists, so any changes made by one person are immediately visible to the other. It is also more complex.

A new file created by one person will automatically appear in all the shared subdirectories.

UNIX INode:

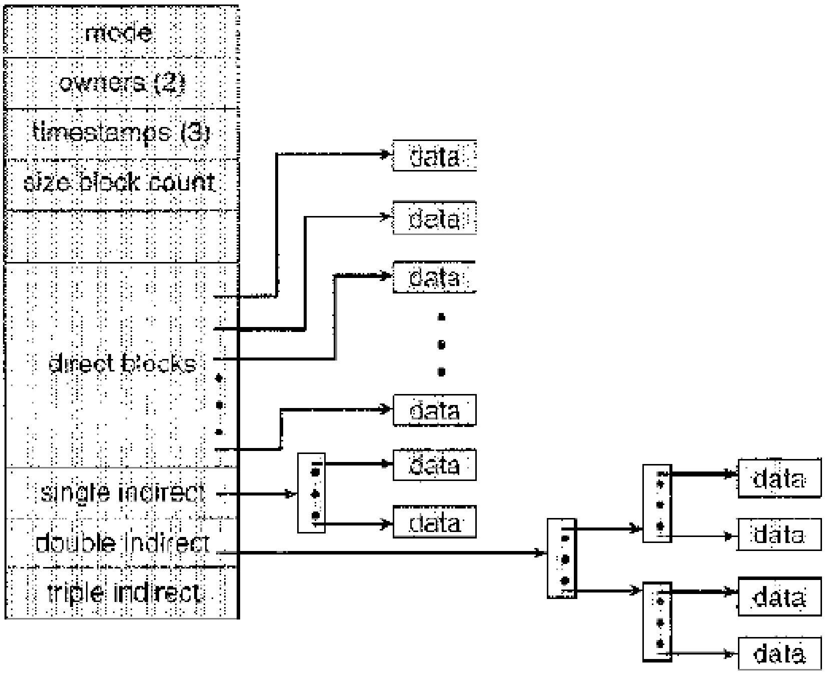


Figure 11.9 The UNIX inode.

File-allocation table (FAT):

An important variation on linked allocation is the use of a file-allocation table (FAT). A section of disk at the beginning of each volume is set aside to contain the table. The table has one entry for each disk block and is indexed by block number.

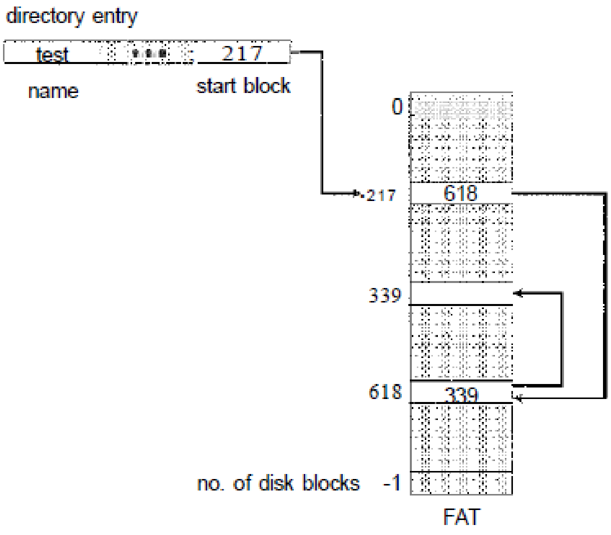


Figure 11.7 File-allocation table.

The directory entry contains the block number of the first block of the file. The table entry indexed by that block number contains the block number of the next block in the file. This chain continues until the last block, which has a special end-of-file value as the table entry. The FAT allocation scheme can result in a significant number of disk head seeks, unless the FAT is cached. A benefit is that **random-access time** is improved, because the disk head can find the location of any block by reading the information in the FAT.

Disk Scheduling Algorithm:

FCFS Scheduling: This algorithm is intrinsically fair, but it generally does not provide the fastest service.

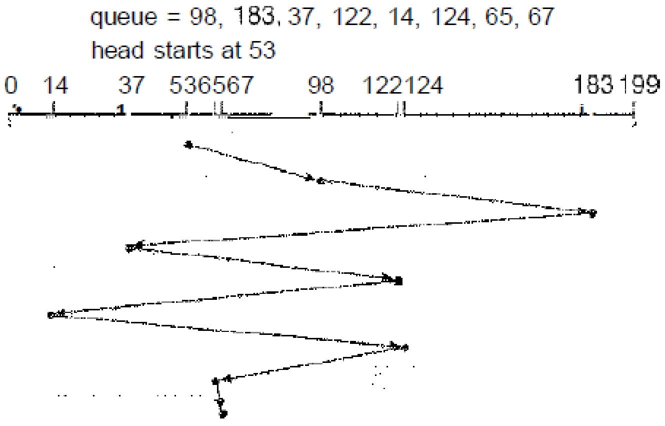


Figure 12.4 FCFS disk scheduling.

If the disk head is initially at cylinder 53, it will first move from 53 to 98, then to 183, 37, 122, 14, 124, 65, and finally to 67, for a total head movement of 640 cylinders.

SSTF Scheduling:

The SSTF algorithm selects the request with the minimum seek time from the current head position.

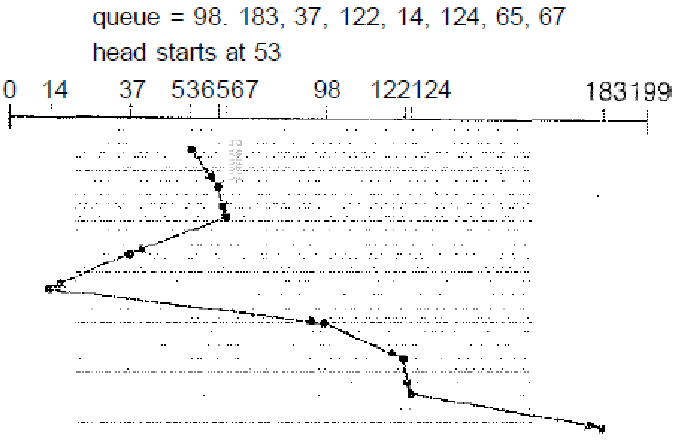


Figure 12.5 SSTF disk scheduling.

Total movements – 236. It may cause starvation to some requests. Although the SSTF algorithm is a substantial improvement over the FCFS algorithm, **it is not optimal**, we even further can improve it.

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.

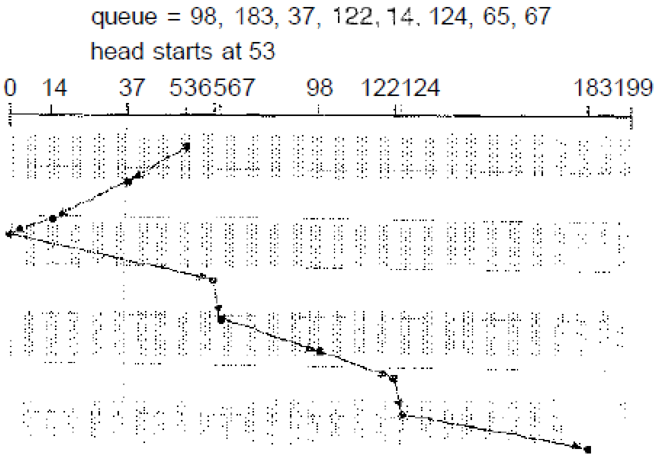


Figure 12.6 SCAN disk scheduling.

Initially, head is moving towards 0. Before applying SCAN to schedule the requests on cylinders 98,183, 37,122,14, 124, 65, and 67, we need to know the direction of head movement in addition to the head's current position (53). If the disk arm is moving toward 0, the head will service 37 and then 14. At cylinder 0, the arm will reverse and will move toward the other end of the disk, servicing the requests at 65, 67, 98, 122, 124, and 183.

C-SCAN Scheduling:

Circular SCAN (C-SCAN) scheduling is a variant of SCAN designed to provide a more uniform wait time.

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**.

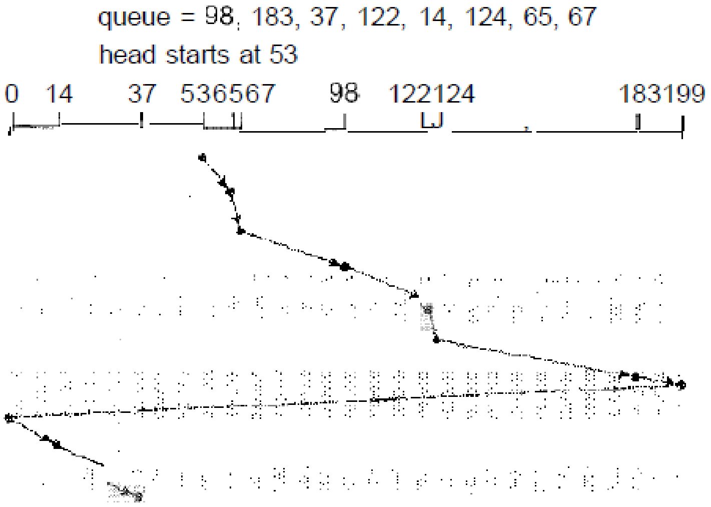


Figure 12.7 C-SCAN disk scheduling.

We’re counting the jump from 199 to 0 (vani notes + other notes).

LOOK Scheduling

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.

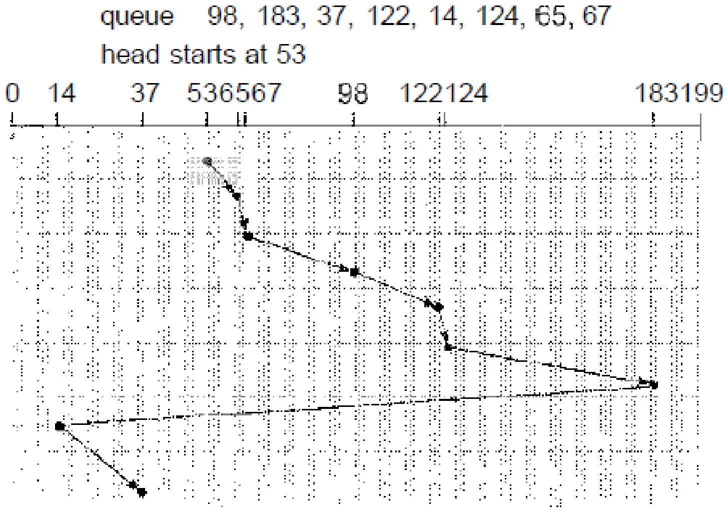


Figure 12.8 C-LOOK disk scheduling.

LOOK: It's similar to the SCAN disk scheduling algorithm except the difference that the disk arm in spite of going to the end of the disk goes only to the last request to be serviced in front of the head and then reverses its direction from there only. Thus, it prevents the extra delay which occurred due to unnecessary traversal to the end of the disk. (geeks for geeks).