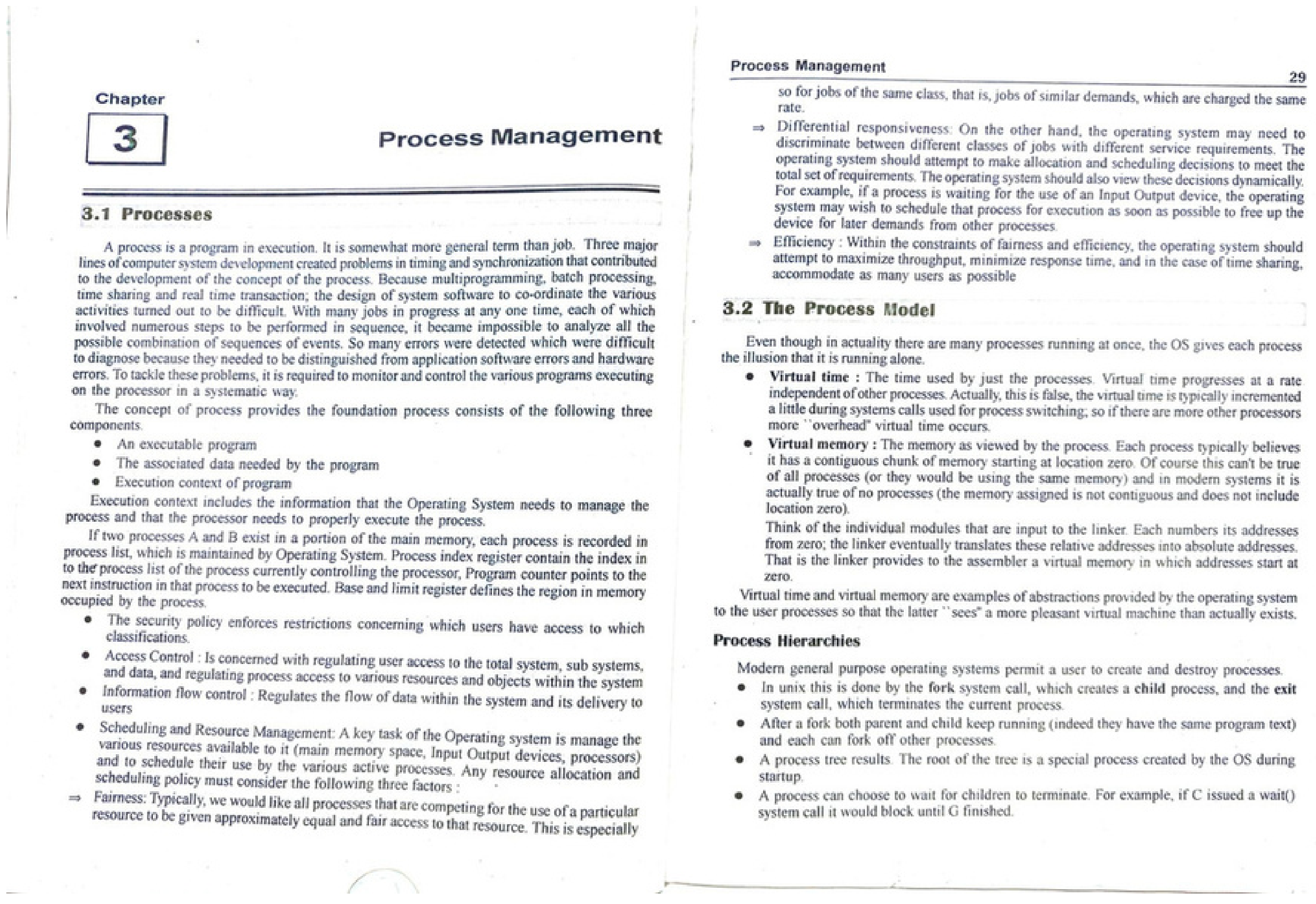
Unit-II

Process Management : Process definition, process control, initializing Operating System, Process Address Spaces Process Abstraction resource Abstraction and Process Hierarchy. Scheduling Mechanisms, Partitioning a process into small processes Non-preemptive strategies (first come-first served, shortest job next, priority scheduling deadline scheduling), Preemptive strategies (Round Robin, two quesues, multiple level queues). Basic Synchronization principles : Interactive processes coordinating processes, Semaphores, Shared memory multiprocessors, AND Synchronization, Inter process communication, inter process messages, mailboxes.

Deadlocks, Resource Status Modeling Handling deadlocks, deadlock detection and resolution deadlock avoidance.



Chapter

3 Process Management

3.1 Processes

A process is a program in execution. It is somewhat more general term than job. Three major lines of computer system development created problems in timing and synchronization that contributed to the development of the concept of the process. Because multiprogramming, batch processing, time sharing and real time transaction; the design of system software to co-ordinate the various activities turned out to be difficult. With many jobs in progress at any one time, each of which involved numerous steps to be performed in sequence, it became impossible to analyze all the possible combination of sequences of events. So many errors were detected which were difficult to diagnose because they needed to be distinguished from application software errors and hardware errors. To tackle these problems, it is required to monitor and control the various programs executing on the processor in a systematic way.

The concept of process provides the foundation process consists of the following three components.

- An executable program
- The associated data needed by the program
- Execution context of program

Execution context includes the information that the Operating System needs to manage the process and that the processor needs to properly execute the process.

If two processes A and B exist in a portion of the main memory, each process is recorded in process list, which is maintained by Operating System. Process index register contain the index in to the process list of the process currently controlling the processor, Program counter points to the next instruction in that process to be executed. Base and limit register defines the region in memory occupied by the process.

- The security policy enforces restrictions concerning which users have access to which classifications.
- Access Control : Is concerned with regulating user access to the total system, sub systems, and data, and regulating process access to various resources and objects within the system
- Information flow control : Regulates the flow of data within the system and its delivery to users
- Scheduling and Resource Management: A key task of the Operating system is manage the various resources available to it (main memory space, Input Output devices, processors) and to schedule their use by the various active processes. Any resource allocation and scheduling policy must consider the following three factors :

⇒ Fairness: Typically, we would like all processes that are competing for the use of a particular resource to be given approximately equal and fair access to that resource. This is especially

so for jobs of the same class, that is, jobs of similar demands, which are charged the same rate.

⇒ Differential responsiveness: On the other hand, the operating system may need to discriminate between different classes of jobs with different service requirements. The operating system should attempt to make allocation and scheduling decisions to meet the total set of requirements. The operating system should also view these decisions dynamically. For example, if a process is waiting for the use of an Input Output device, the operating system may wish to schedule that process for execution as soon as possible to free up the device for later demands from other processes.

⇒ Efficiency : Within the constraints of fairness and efficiency, the operating system should attempt to maximize throughput, minimize response time, and in the case of time sharing, accommodate as many users as possible

3.2 The Process Model

Even though in actuality there are many processes running at once, the OS gives each process the illusion that it is running alone.

- **Virtual time :** The time used by just the processes. Virtual time progresses at a rate independent of other processes. Actually, this is false, the virtual time is typically incremented a little during systems calls used for process switching; so if there are more other processors more ``overhead" virtual time occurs.
- **Virtual memory :** The memory as viewed by the process. Each process typically believes it has a contiguous chunk of memory starting at location zero. Of course this can't be true of all processes (or they would be using the same memory) and in modern systems it is actually true of no processes (the memory assigned is not contiguous and does not include location zero).

  Think of the individual modules that are input to the linker. Each numbers its addresses from zero; the linker eventually translates these relative addresses into absolute addresses. That is the linker provides to the assembler a virtual memory in which addresses start at zero.

Virtual time and virtual memory are examples of abstractions provided by the operating system to the user processes so that the latter ``sees" a more pleasant virtual machine than actually exists.

Process Hierarchies

Modern general purpose operating systems permit a user to create and destroy processes.

- In unix this is done by the **fork** system call, which creates a **child** process, and the **exit** system call, which terminates the current process.
- After a fork both parent and child keep running (indeed they have the same program text) and each can fork off other processes.
- A process tree results. The root of the tree is a special process created by the OS during startup.
- A process can choose to wait for children to terminate. For example, if C issued a wait() system call it would block until G finished.

Old or primitive operating system like MS-DOS are not multiprogrammed so when one process starts another, the first process is automatically blocked and waits until the second is finished.

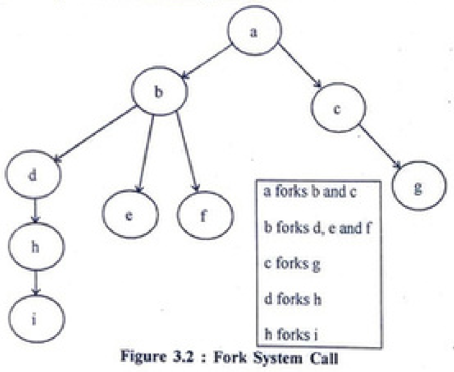
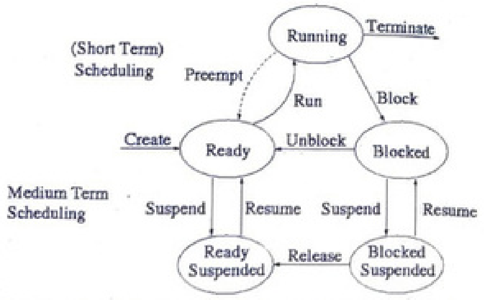


Figure 3.2 : Fork System Call

3.3 Process states and Transitions



Unblock is done by another task (a.k.a wakeup, release, allocate, V)
Block a.k.a sleep request, P

Figure 3.2 : Process Transition

Process can have one of the following fives states a time-

1. **Create :** The process is being created.
2. **Ready :** The process is waiting to be assigned to a processor. Ready processes are waiting to have the processor allocated to them by the operating system so that they can sum.
3. **Running :** Process instruction are being executed.
4. **Waiting :** The process is waiting for some event to occur.
5. **Terminated :** The process has finished execution.

The above diagram contains a great deal of information.

- Consider a running process P that issues an Input Output request
    - The process blocks
    - At some later point, a disk interrupt occurs and the driver detects that P's request is satisfied.
    - P is unblocked, i.e. is moved from blocked to ready
    - At some later time the operating system looks for a ready job to run and picks P.
- A preemptive scheduler has the dotted line preempt;
  A non-preemptive scheduler doesn't.
- The number of processes changes only for two arcs: create and terminate.
- Suspend and resume are medium term scheduling
    - Done on a longer time scale.
    - Involves memory management as well.
    - Sometimes called two level scheduling.

3.3.1 Process Control Block

For each process there is a Process Control Block, PCB, which stores the following (types of) process-specific information. (Specific details may vary from system to system). A process control block or PCB is data on structure (a table) that holds information about a process.

Each process contains the process control block (PCB). PCB is the data structure used by the operating system. Operating system groups all information that needs about particular process. Fig. shows the process control block.

1. **Pointer :** Pointer points to another process control block. Pointer is used for maintaining the scheduling list.
2. **Process State :** Process state may be new, ready, running, waiting and so on.

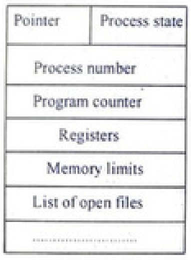


Figure 3.3.1 : Process control block (PCB)

3. **Program Counter :** It indicates the address of the next instruction to be executed for this process. Saved and restored when swapping processes in and out of the CPU.
4. **CPU register:** It indicates general purpose register, stack pointers, index registers and accumulator's etc. number of register and type of register totally depends upon the computer architecture.
5. **Memory Management Information :** This information may include the value of base and limit register. This information is useful for de-allocating the memory when the process terminates.

3.4 Initialization operating system

Initialization is the process of locating and using the defined values for variable data that is used by a computer program. For example, an operating system or application program is installed with default or user-specified values that determine certain aspects of how the system or program is to function.

In many general computing devices, this read only memory defines a BIOS transfers control to the cartridge after doing some preliminary tests to make sure the machinery is working correctly. On other machines like the PC and Mac, BIOS calls a utility from read only memory and lets that check the machinery.

1. Power On self Test
2. Boot Sector
3. Kernel Initialization
4. File System Initialization
5. Plug and Play
6. Hot Socketing

1. Power On self Test : The POST, When you turn on or restart a computer, it begins a POST routine. The POST routine determines the available amount of real memory and verifies the presence of required hardware components, such as the keyboard.

After the computer runs its POST routine, each adapter card with a basic input/output system (BIOS) runs its own POST routine. The computer and adapter card manufacturers determine what appears on the screen during POST processing.

2. Boot Sector : A boot sector or boot block is a region of a hard disk, floppy disk, optical disc, or other data storage device that contains machine code to be loaded into random-access memory (RAM) by a computer system's built-in firmware. The purpose of a boot sector is to allow the boot process of a computer to load a program (usually, but not necessarily, an operating system) stored on the same storage device. The location and size of the boot sector (perhaps corresponding to a logical disk sector) is specified by the design of the computing platform.

3. Kernel Initialization : Once the kernel is fully loaded, the next step in initialization is to set the kernel parameters and options, and add any modules that have been selected in the kernel set-up file. Once the kernel is fully initialized it takes over control of the computer and continues initialization with the file-systems and processes.

4. File System Initialization : The kernel starts up the processes, and loads the file-systems. The main file-system then includes initialization files, which can be used to set up the operating systems environment, and initialize all the services, daemons, and applications.

5. Plug and Play : Plug and Play (PnP) is a capability developed by Microsoft for its Windows 95 and later operating systems that gives users the ability to plug a device into a computer and have the computer recognize that the device is there. Plug and Play (PnP) is a capability developed by Microsoft for its Windows 95 and later operating systems that gives users the ability to plug a device into a computer and have the computer recognize that the device is there. The user doesn't have to tell the computer. In many earlier computer systems, the user was required to explicitly tell the operating system when a new device had been added. Microsoft made Plug and play a selling point for its Windows operating systems. A similar capability had long been built into Macintosh computers.

6. Hot Socketing : A further extension of this concept was developed for the Universal Serial Bus, which allows devices to be hot socketed, or installed while the computer is running. The USB bus contacts the new device and learns from it the necessary information to match it to a driver. This information gets put into the database, and whenever that device is again plugged in, the same driver is found for it. When the device is unsocketted, the driver shuts itself down and removes itself from the list of active devices. It can do this because it can monitor the USB controller to make sure it's device is still attached.

3.5 Process Address Space

"Memory Management," looked at how the kernel manages physical memory. In addition to managing its own memory, the kernel also has to manage the process address spacethe representation of memory given to each user-space process on the system. Linux is a virtual memory operating system, and thus the resource of memory is virtualized among the processes on the system. To an individual process, the view is as if it alone has full access to the system's physical memory. More importantly, the address space of even a single process can be much larger than physical memory. This chapter discusses how the kernel manages the process address space.

The process address space consists of the linear address range presented to each process and, more importantly, the addresses within this space that the process is allowed to use. Eac' process is given a flat 32- or 64-bit address space, with the size depending on the architectur The term "flat" describes the fact that the address space exists in a single range. (As an examp a 32-bit address space extends from the address 0 to 429496729.) Some operating systems provic a segmented address space, with addresses existing not in a single linear range, but instead in multiple segments. Modern virtual memory operating systems generally have a flat memory model and not a segmented one. Normally, this flat address space is unique to each process. A memory address in one process's address space tells nothing of that memory address in another process's address space. Both processes can have different data at the same address in their respective address spaces. Alternatively, processes can elect to share their address space with other processes. We know these processes as threads.

A memory address is a given value within the address space, such as 4021f000. This particular value identifies a specific byte in a process's 32-bit address space. The interesting part of the address space is the intervals of memory addresses, such as 08048000-0804c000, that the process has permission to access. These intervals of legal addresses are called memory areas. The process, through the kernel, can dynamically add and remove memory areas to its address space.

The process can access a memory address only in a valid memory area. Memory areas have associated permissions, such as readable, writable, and executable, that the associated process must respect. If a process accesses a memory address not in a valid memory area, or if it accesses a valid area in an invalid manner, the kernel kills the process with the dreaded "Segmentation Fault" message.

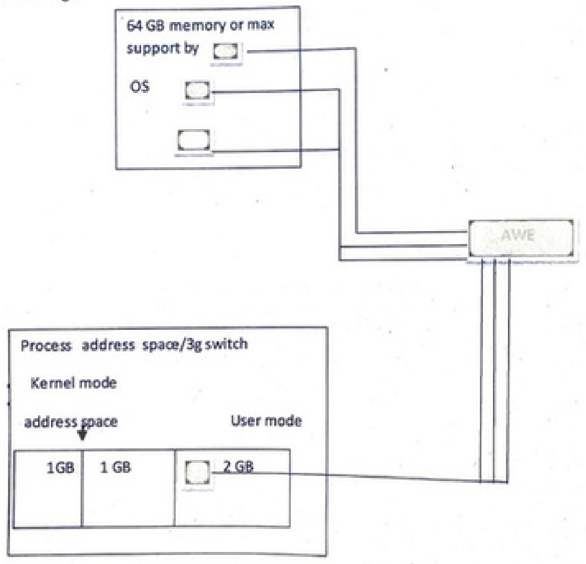

64 GB memory or max support by
OS
AWE
Process address space/3g switch
Kernel mode
address space
User mode
1GB
1 GB
2 GB

Figure 3.5 : Process address space

All 32-bit applications have a 4-gigabyte (GB) process address space (32-bit addresses can map a maximum of 4 GB of memory). Microsoft Windows operating systems provide applications with access to 2 GB of process address space, specifically known as user mode virtual address space. All threads owned by an application share the same user mode virtual address space. The

remaining 2 GB are reserved for the operating system (also known as kernel mode address space). All operating system editions starting with Windows 2000 Server, including Windows Server 2003, have a boot.ini switch that can provide applications with access to 3 GB of process address space, limiting the kernel mode address space to 1 GB.

This feature will be removed in the next version of Microsoft SQL Server. Do not use this feature in new development work, and modify applications that currently use this feature as soon as possible.

Address Windowing Extensions (AWE) extend the capabilities of 32-bit applications by allowing access to as much physical memory as the operating system supports. AWE accomplishes this by mapping a subset of up to 64 GB into the user address space. Mapping between the application buffer pool and AWE-mapped memory is handled through manipulation of the Windows virtual memory tables.

To enable support for 3 GB of user mode process space, you must add the /3gb parameter to the boot.ini file and reboot the computer, allowing the /3gbparameter to take effect. Setting this parameter allows user application threads to address 3 GB of process address space, and reserves 1 GB of process address space for the operating system.

3.6 Threads

- A thread is a basic unit of CPU utilization, consisting of a program counter, a stack, and a set of registers, (and a thread ID.)
- Traditional (heavyweight) processes have a single thread of control - There is one program counter, and one sequence of instructions that can be carried out at any given time.
- As shown in Figure, multi-threaded applications have multiple threads within a single process, each having their own program counter, stack and set of registers, but sharing common code, data, and certain structures such as open files.

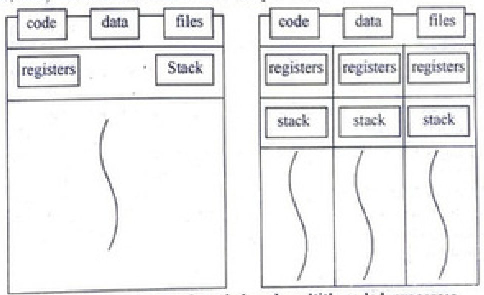


Figure 3.6 : - Single-threaded and multithreaded processes

Difference between Process and Thread

S.N.	Process	Thread
1	Process is heavy weight or resource intensive.	Thread is light weight taking lesser resources than a process.
2	Process switching needs interaction with operating system.	Thread switching does not need to interact with operating system.
3	In multiple processing environments each process executes the same code but has its own memory and file resources.	All threads can share same set of open files, child processes.
4	If one process is blocked then no other process can execute until the first process is unblocked.	While one thread is blocked and waiting, second thread in the same task can run.
5	Multiple processes without using threads use more resources.	Multiple threaded processes use fewer resources.
6	In multiple processes each process operates independently of the others.	One thread can read, write or change another thread's data.

Benefits

There are four major categories of benefits to multi-threading:

1. Responsiveness - One thread may provide rapid response while other threads are blocked or slowed down doing intensive calculations.
2. Resource sharing - By default threads share common code, data, and other resources, which allows multiple tasks to be performed simultaneously in a single address space.
3. Economy - Creating and managing threads (and context switches between them) is much faster than performing the same tasks for processes.
4. Scalability, i.e. Utilization of multiprocessor architectures - A single threaded process can only run on one CPU, no matter how many may be available, whereas the execution of a multi-threaded application may be split amongst available processors. (Note that single threaded processes can still benefit from multi-processor architectures when there are multiple processes contending for the CPU, i.e. when the load average is above some certain threshold.)

3.7 Types of Thread

Threads are implemented in following two ways

- User Level Threads -- User managed threads
- Kernel Level Threads -- Operating System managed threads acting on kernel, an operating system core.

User Level Threads

In this case, application manages thread management kernel is not aware of the existence of threads. The thread library contains code for creating and destroying threads, for passing message and data between threads, for scheduling thread execution and for saving and restoring thread contexts. The application begins with a single thread and begins running in that thread.

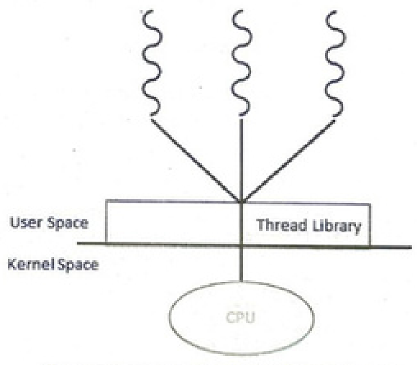


Figure 3.7 : User level and Kernel level threads

ADVANTAGES

- Thread switching does not require Kernel mode privileges.
- User level thread can run on any operating system.
- Scheduling can be application specific in the user level thread.
- User level threads are fast to create and manage.

DISADVANTAGES

- In a typical operating system, most system calls are blocking.
- Multithreaded application cannot take advantage of multiprocessing.

Kernel Level Threads

In this case, thread management done by the Kernel. There is no thread management code in the application area. Kernel threads are supported directly by the operating system. Any application can be programmed to be multithreaded. All of the threads within an application are supported within a single process.

The Kernel maintains context information for the process as a whole and for individuals threads within the process. Scheduling by the Kernel is done on a thread basis. The Kernel performs thread creation, scheduling and management in Kernel space. Kernel threads are generally slower to create and manage than the user threads.

ADVANTAGES

- Kernel can simultaneously schedule multiple threads from the same process on multiple processes.
- If one thread in a process is blocked, the Kernel can schedule another thread of the same process.
- Kernel routines themselves can multithreaded.

DISADVANTAGES

- Kernel threads are generally slower to create and manage than the user threads.
- Transfer of control from one thread to another within same process requires a mode switch to the Kernel.

3.8 Multithreading Models

Some operating system provide a combined user level thread and Kernel level thread facility. Solaris is a good example of this combined approach. In a combined system, multiple threads within the same application can run in parallel on multiple processors and a blocking system call need not block the entire process. Multithreading models are three types

- Many to many relationship.
- Many to one relationship.
- One to one relationship.

3.8.1 Many to Many Model

In this model, many user level threads multiplexes to the Kernel thread of smaller or equal numbers. The number of Kernel threads may be specific to either a particular application or a particular machine.

Following diagram shows the many to many model. In this model, developers can create as many user threads as necessary and the corresponding Kernel threads can run in parallels on a multiprocessor.

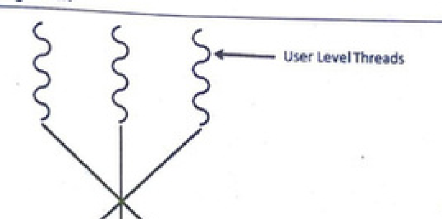


Figure 3.8.1 : Many to Many Model

Many to One Model

Many to one model maps many user level threads to one Kernel level thread. Thread management is done in user space. When thread makes a blocking system call, the entire process will be blocks. Only one thread can access the Kernel at a time,so multiple threads are unable to run in parallel on multiprocessors.

If the user level thread libraries are implemented in the operating system in such a way that system does not support them then Kernel threads use the many to one relationship modes.

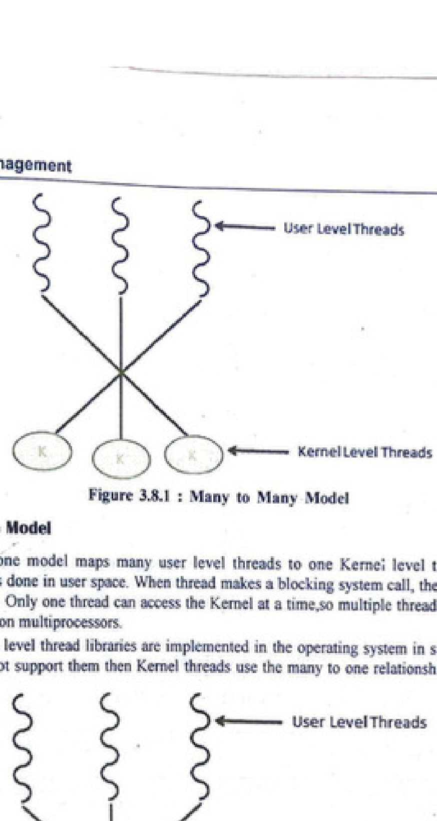


Figure 3.8.2 : Many to One Model

3.8.3 One to One Model

There is one to one relationship of user level thread to the kernel level thread.This model provides more concurrency than the many to one model. It also another thread to run when a thread makes a blocking system call. It support multiple thread to execute in parallel on microprocessors. Disadvantage of this model is that creating user thread requires the corresponding Kernel thread. OS/2, windows NT and windows 2000 use one to one relationship model.

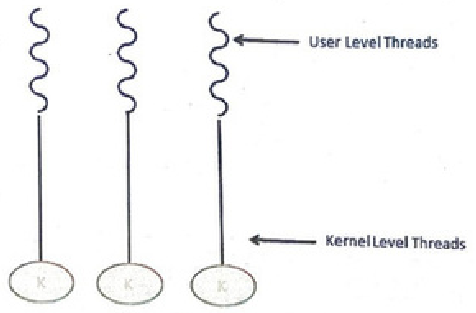


Figure 3.8.2 : One to One Model

3.9 Difference between User Level & Kernel Level Thread

S.N.	User Level Threads	Kernel Level Thread
1	User level threads are faster to create and manage.	Kernel level threads are slower to create and manage.
2	Implementation is by a thread library at the user level.	Operating system supports creation of Kernel threads.
3	User level thread is generic and can run on any operating system.	Kernel level thread is specific to the operating system.
4	Multi-threaded application cannot take advantage of multiprocessing.	Kernel routines themselves can be multithreaded.

Exercise

Part I (Very Short Answer)

1. What is a thread?
2. Write two advantages of multi-threading.
3. What is a PCB?
4. What are different process states.

Part II (Short Answer)

1. Differentiate between a program and a process.
2. Give any three differences between a process and a thread.
3. What are the reasons of process suspension?
4. What are the features of process management?

Part III (Long Answer)

1. Explain the concept of process.
2. Compare user level and kernel level threads.

Chapter

4.

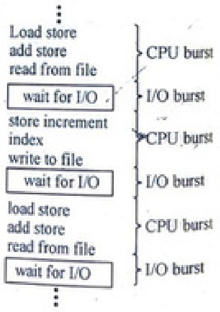
Scheduling

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 this chapter, we introduce basic CPU-scheduling concepts and present several CPU-scheduling algorithms. We also consider the problem of selecting an algorithm for a particular system.

In Chapter 3, we introduced threads to the process model. On operating systems that support them, it is kernel-level threads-not processes-that are in fact being scheduled by the operating system. However, the terms process scheduling and thread scheduling are often used interchangeably. In this chapter, we use process scheduling when discussing general scheduling concepts and thread scheduling to refer to thread-specific ideas.

4.1 Concepts

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. The idea is relatively simple. A process is executed until it must wait, typically for the completion of some Input Output request. In a simple computer system, the CPU then just sits idle. All this waiting time is wasted; no useful work is accomplished. With multiprogramming, we try to use this time productively. Several processes are kept in memory at one time.



When one process has to wait, the operating system takes the CPU away from that process and gives the CPU to another process. This pattern continues. Every time one process has to wait, another process can take over use of the CPU.

Scheduling of this kind is a fundamental operating-system function. Almost all computer resources are scheduled before use. The CPU is, of course, one of the primary computer resources. Thus, its scheduling is central to operating-system design.

4.1.1 CPU-i/O Burst Cycle

The success of CPU scheduling depends on an observed property of processes: 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 Input Output burst, which is followed by another CPU burst, then another Input Output burst, and so on. Eventually, the final CPU burst ends with a system request to terminate execution.

The durations of CPU bursts have been measured extensively. Although they vary greatly from process to process and from computer to computel~ they tend to have a frequency curve. The curve is generally characterized as exponential or hyper-exponential, with a large number of short CPU bursts and a small number of long CPU bursts. An Input Output-bound program typically has many short CPU bursts. A CPU-bound program might have a few long CPU bursts.

4.1.2 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 that the ready queue is not necessarily a first-in, first-out (FIFO) queue. As we shall see when we consider the various scheduling algorithms, a ready queue can be implemented as a FIFO queue, a priority queue, a tree, or simply an unordered linked list. Conceptually, howevel~ all the processes in the ready queue are lined up waiting for a chance to run on the CPU. The records in the queues are generally process control blocks (PCBs) of the processes.

In a complex operating system three types of schedulers :

1. Long-term scheduler
2. Short-term scheduler
3. Medium-term scheduler

Comparision between scheduler

S.No.	Long term scheduler	Short term Scheduler	Medium term Scheduler
1.	It is a job scheduler	It is a CPU scheduler	It is a process swapping scheduler
2.	Speed is lesser than short term scheduler	Speed is fastest among other two	Speed is in between both short & Long term scheduler
3.	It controls the degree of multi programming	It provides lesser control over degree of multi programming	It reduce the degree of multi programming

4.2 Preemptive and non Preemptive 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/0 request or an invocation of wait for the termination of one of the child processes)

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 example, 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. There is a choice, however, for situations 2 and 3.

When scheduling takes place only under circumstances 1 and 4, we say that the scheduling scheme is non-preemptive or cooperative; otherwise, it is preemptive.

Under non-preemptive 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. This scheduling method was used by Microsoft Windows 3.x; Windows 95 introduced preemptive scheduling, and all subsequent versions of Windows operating systems have used preemptive scheduling.

The Mac OS X operating system for the Macintosh also uses preemptive scheduling; previous versions of the Macintosh operating system relied on cooperative scheduling. Cooperative scheduling is the only method that can be used on certain hardware platforms, because it does not require the special hardware (for example, a timer) needed for preemptive scheduling.

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.

Comparis on between Preemptive and Non Preemptive scheduling

S.No.	Preemptive	Non Preemptive
1.	A scheduling discipline is preemptive, if the CPU can be taken away from a process after being allocated.	A scheduling discipline is non preemptive, if once a process has been given the CPU, the CPU cannot be taken away from that process.
2.	Higher priority jobs are processed before the lower priority jobs	Short jobs are made to wait by longer jobs.
3.	If the CPU has been allocated to certain process, it can be snatched from this process any time either due to time constraint or due to priority reason	Once the CPU has been allocated to a process, the process keeps the CPU until it release the CPU either by terminating or by switching to the waiting state.
4.	It is costly as compared to non preemptive scheduling	It is not costly as compared to preemptive scheduling

4.3 Scheduling Mechanism

Different CPU-scheduling algorithms have different properties, and the choice of a particular

algorithm may favor one class of processes over another. In choosing which algorithm to use in a particular situation, we must consider the properties of the various algorithms.

Many criteria have been suggested for comparing CPU-scheduling algorithms. Which characteristics are used for comparison can make a substantial difference in which algorithm is judged to be best. The criteria include the following:

- **CPU utilization :** We want to keep the CPU as busy as possible. Conceptually, CPU utilization can range from 0 to 100 percent. In a real system, it should range from 40 percent (for a lightly loaded system) to 90 percent (for a heavily used system).
- **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. For long processes, this rate may be one process per hour; for short transactions, it may be ten processes per second.
- **Turnaround time. :** From the point of view of a particular process, 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/0; it affects only the an1 ount 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 :** In an interactive system, turnaround time may not be the best criterion. 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 since it takes to start responding, not the time it takes to output the response. The turnaround time is generally limited by the speed of the output device.

It is desirable to maximize CPU utilization and throughput and to minimize turnaround time, waiting time, and response time. In most cases, we optimize the average measure. However, under some circumstances, it is desirable to optimize the minimum or maximum values rather than the average. For example, to guarantee that all users get good service, we may want to minimize the maximum response time.

Investigators have suggested that, for interactive systems (such as timesharing systems), it is more important to minimize the variance in the response time than to minimize the average response time. 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. Howevel – little work has been done on CPU-scheduling algorithms that minimize variance.

As we discuss various CPU-scheduling algorithms in the following section, we illustrate their operation. An accurate illustration should involve many processes, each a sequence of several hundred CPU bursts and Input Output bursts. For simplicity, though, we consider only one CPU burst (in milliseconds) per process in our examples. Our measure of comparison is the average waiting time.

4.3.1 First-Come, First-Served Scheduling

By far the simplest CPU-scheduling algorithm is the first-come, first-served (FCFS) scheduling algorithm. With this scheme, the process that requests the CPU first is allocated the CPU first. 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. The running process is then removed from the queue. The code for FCFS scheduling is simple to write and understand. On the negative side, the average waiting time under the FCFS policy is often quite long. Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds:

Process	Burst Time
P1	24
P2	3
P3	3

If the processes arrive in the order P1, P2, P3, and are served in FCFS order. The waiting time is 0 milliseconds for process P1, 24 milliseconds for process P2 , and 27 milliseconds for process P3 . Thus, the average waiting time is (0 + 24 + 27)/3 = 17 neilliseconds. If the processes arrive in the order P2, P3 , P1, However.

The average waiting time is now (6 + 0 + 3)/3 = 3 milliseconds. This reduction is substantial. Thus, the average waiting time under an FCFS policy is generally not minimal and may vary substantially if the processes CPU burst times vary greatly.

In addition, consider the performance of FCFS scheduling in a dynamic situation. Assume we have one CPU-bound process and many Input Output-bound processes. As the processes flow around the system, the following scenario may result.

The CPU-bound process will get and hold the CPU. During this time, all the other processes will finish their I/0 and will move into the ready queue, waiting for the CPU. While the processes wait in the ready queue, the I/0 devices are idle. Eventually, the CPU-bound process finishes its CPU burst and moves to an I/0 device. All the Input Output-bound processes, which have short CPU bursts, execute quickly and move back to the I/0 queues.

At this point, the CPU sits idle. The CPU-bound process will then move back to the ready queue and be allocated the CPU. Again, all the I/0 processes end up waiting in the ready queue until the CPU-bound process is done. There is a convoy effect as all the other processes wait for the one big process to get off the CPU. This effect results in lower CPU and device utilization than might be possible if the shorter processes were allowed to go first.

Note also that the FCFS scheduling algorithm is non-preemptive. Once the CPU has been allocated to a process, that process keeps the CPU.until it releases the process, either by terminating or by requesting I/0. The FCFS algorithm is thus particularly troublesome for time-sharing systems, where it is important that each user get a share of the CPU at regular intervals. It would be disastrous to allow one process to keep the CPU for an extended period.

4.3.2 Shortest Job Next Scheduling

A different approach to CPU scheduling is the shortest-job-first (SJF) scheduling algorithm. 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. Note that 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. We use the term SJF because m.ost people and textbooks use this term to refer to this type of scheduling. As an example of SJF scheduling, consider the following set of processes, with the length of the CPU burst given in milliseconds:

Process	Burst Time
P1	6
P2	8
P3	7
P4	3

The waiting time is 3 milliseconds for process P1, 16 milliseconds for process P2, 9 milliseconds for process P3, and 0 milliseconds for process P4 . Thus, the average waiting time is (3 + 16 + 9 + 0) I 4 = 7 milliseconds. By comparison, if we were using the FCFS scheduling scheme, the average waiting time would be 10.25 milliseconds.

The SJF scheduling algorithm is provably optimal, in that it gives the minimum average waiting time for a given set of processes. Moving a short process before long one decrease the waiting time of the short process more than it increases the waiting time of the long process. Consequently, the average waiting time decreases.

The real difficulty with the SJF algorithm knows the length of the next CPU request. For long-term (job) scheduling in a batch system, we can use as the length the process time limit that a user specifies when he submits the job. Thus, users are motivated to estimate the process time limit accurately, since a lower value may mean faster response. (Too low a value will cause a time-limit-exceeded error and require resubmission.) SJF scheduling is used frequently in long-term scheduling.

Although the SJF algorithm is optimal, it cannot be implemented at the level of short-term CPU scheduling. With short-term 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. We expect that the next CPU burst will be similar in length to the previous ones. By computing an approximation of the length of the next CPU burst, we can pick the process with the shortest predicted CPU burst.

4.3.3 Priority Scheduling

The SJF algorithm is a special case of the general priority scheduling algorithm. A priority is associated with each process, and the CPU is allocated to the process with the highest priority.

Equal-priority processes are scheduled in FCFS order.

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.

Note that we discuss scheduling in terms of high priority and low priority. Priorities are generally indicated by some fixed range of numbers, such as 0 to 7 or 0 to 4,095. However, there is no general agreement on whether 0 is the highest or lowest priority. Some systems use low numbers to represent low priority; others use low numbers for high priority. This difference can lead to confusion. In this text, we assume that low numbers represent high priority.

As an example, consider the following set of processes, assumed to have arrived at time 0 in the order P1, P2, · · ·, Ps, with the length of the CPU burst given in milliseconds:

Process	Burst Time	Priority
P1	10	3
P2	1	1
P3	2	4
P4	1	5
P5	5	2

The average waiting time is 8.2 milliseconds. Priorities can be defined either internally or externally. Internally defined priorities use some measurable quantity or quantities to compute the priority of a process. For example, time limits, memory requirements, the number of open files, and the ratio of average I/0 burst to average CPU burst have been used in computing priorities. External priorities are set by criteria outside the operating system, such as the importance of the process, the type and amount of funds being paid for computer use, the department sponsoring the work, and other, often political factors.

Priority scheduling can be either preemptive or non-preemptive. When a process arrives at the ready queue, its priority is compared with the priority of the currently running process. A preemptive priority scheduling algorithm will preempt the CPU if the priority of the newly arrived process is higher than the priority of the currently running process. A non-preemptive priority scheduling algorithm will simply put the new process at the head of the ready queue.

A major problem with priority scheduling algorithms is indefinite blocking, or starvation. A process that is ready to run but waiting for the CPU can be considered blocked. A priority scheduling algorithm can leave some low priority processes waiting indefinitely. In a heavily loaded computer system, a steady stream of higher-priority processes can prevent a low-priority process from ever getting the CPU. Generally, one of two things will happen. Either the process will eventually be run (at 2 A.M. Sunday, when the system is finally lightly loaded), or the computer system will eventually crash and lose all unfinished low-priority processes.

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. For example, if priorities range from 127 (low) to 0 (high), we could increase the priority of a waiting process by 1 every 15 minutes. Eventually, even a process with an initial priority of 127

would have the highest priority in the system and would be executed. In fact, it would take no more than 32 hours for a priority-127 process to age to a priority-0 process.

4.3.4 Round-Robin Scheduling

The round-robin (RR) scheduling algorithm is designed especially for timesharing systems. It is similar to FCFS scheduling, but preemption is added to enable the system to switch between processes. A small unit of time, called a time quantum or time slice, is defined. A time quantum is generally fronc 10 to 100 milliseconds in length. 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 o£ 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.

One of two things will then happen. The process may have a CPU burst of less than 1 time quantum. In this case, the process itself will release the CPU voluntarily. The scheduler will then proceed to the next process in the ready queue. Otherwise, if the CPU burst of the currently running process is longer than 1 time quantum, the timer will go off and will cause an interrupt to the operating system. A context switch will be executed, and the process will be put at the tail o£ the ready queue. The CPU scheduler will then select the next process in the ready queue.

The average waiting time under the RR policy is often long. Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds:

Process	Burst Time
P1	24
P2	3
P3	3

If we use a time quantum of 4 milliseconds, then process P1 gets the first 4 milliseconds. Since it requires another 20 milliseconds, it is preempted after the first time quantum, and the CPU is given to the next process in the queue, process P2 . Process P2 does not need 4 milliseconds, sc it quits before its time quantum expires. The CPU is then given to the next process, process P3 Once each process has received 1 time quantum, the CPU is returned to process P1 for ε additional time quantum.

Let's calculate the average waiting time for the above schedule. P1 waits for 6 millisconds (10- 4), P2 waits for 4 millisconds, and P3 waits for 7 millisconds. Thus, the average waiting time is 17/3 = 5.66 milliseconds. 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). If a process's CPU burst exceeds 1 time quantum, that process is preempted and is p11t back in the ready queue. The RR scheduling algorithm is thus preemptive.

If there are n. processes in the ready queue and the time quantum is q, then each process gets 1 In of the CPU time in chunks of at most q time units. Each process must wait no longer than (11

- 1) x q time units until its next time quantum. For example, with five processes and a time quantum of 20 milliseconds, each process will get up to 20 milliseconds every 100 milliseconds.

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. In contrast, 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 11 processes has its own processor running at 1 I 11 the speed of the real processor. This approach was used in Control Data Corporation (CDC) hardware to implement ten peripheral processors with only one set of hardware and ten sets of registers. The hardware executes one instruction for one set of registers, then goes on to the next. This cycle continues, resulting in ten slow processors rather than one fast one. (Actually, since the processor was much faster than memory and each instruction referenced memory, the processors were not much slower than ten real processors would have been.) In software, we need also to consider the effect of context switching on the performance of RR scheduling. Assume, for example, 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 quantum is 6 time units, however, the process requires 2 quanta, resulting in a context switch. If the time quantum is 1 time unit, then nine context switches will occur, slowing the execution of the process accordingly.

Thus, we want the time quantum to be large with respect to the context switch time. If the context-switch time is approximately 10 percent of the time quantum, then about 10 percent of the CPU time will be spent in context switching. In practice, most modern systems have time quanta ranging from 10 to 100 milliseconds. The time required for a context switch is typically less than 10 microseconds; thus, the context-switch time is a small fraction of the time quantum.

Turnaround time also depends on the size of the time quantum. As we can see from Figure, the average turnaround time of a set of processes does not necessarily improve as the time-quantum size increases. In general, the average turnaround time can be improved if most processes finish their next CPU burst in a single time quantum. For example, given three processes of 10 time units each and a quantum of 1 time unit, the average turnaround time is 29. If the time quantum is 10, however, the average turnaround time drops to 20. If context-switch time is added in, the average turnaround time increases even more for a smaller time quantum, since more context switches are required.

Although the time quantum should be large compared with the context switch time, it should not be too large. If the time quantum is too large, RR scheduling degenerates to an FCFS policy. A rule of thumb is that 80 percent of the CPU bursts should be shorter than the time quantum.

4.3.5 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. For example, separate queues might be used for foreground and background processes. The foreground queue might be scheduled by an RR algorithm, while the background queue is scheduled by an FCFS algorithm.

In addition, there must be scheduling among the queues, which is commonly implemented as fixed-priority preemptive scheduling. For example, the foreground queue may have absolute priority over the background queue. Let's look at an example of a multilevel queue scheduling algorithm with five queues, listed below in order of priority:

- System processes
- Interactive processes
- Interactive editing processes
- Batch processes
- 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 preempted.

Another possibility is to time-slice among the queues. Here, each queue gets a certain portion of the CPU time, which it can then schedule among its various processes. For instance, in the foreground-background queue example, the foreground queue can be given 80 percent of the CPU time for RR scheduling among its processes, whereas the background queue receives 20 percent of the CPU to give to its processes on an FCFS basis.

Exercise

Part I (Very Short Answer)

1. What is CPU utilization?
2. What is turn-around time?
3. Write about CPU-Input Output burst cycle.
4. Write about CPU Scheduler.

Part II (Short Answer)

5. Differentiate between pre-emptive and non pre-emptive process scheduling.
6. Compare Round robin and SJF scheduling in terms of turn-around time.
7. Write about multi level Queue scheduling.

Part III (Long Answer)

8. Explain the FCFS Scheduling with example.
9. Explain priority scheduling in detail.
10. On a System using Round Robin Scheduling. What would be the effect of including one process twice in the list of prcessess.

■■■

Chapter

5 Process Synchronization

5.1 Interactive Process and Coordinating Process

A coordinating process is one that can affect or be affected by other processes executing in the system. Coordinating 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. Concurrent access to shared data may result in data inconsistency, however. In this chapter, we discuss various mechanisms to ensure the orderly execution of cooperating processes that share a logical address space, so that data consistency is maintained.

we developed a model of a system consisting of cooperating sequential processes or threads, all running asynchronously and possibly sharing data. We illustrated this model with the producer-consumer problem, which is representative of operating systems. We described how a bounded buffer could be used to enable processes to share memory.

Let's return to our consideration of the bounded buffer. As we pointed out, our original solution allowed 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) {
I* produce an item in nextProduced *I
while (counter == BUFFER_SIZE)
; I* do nothing *I
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)
; I* do nothing *I
nextConsumed = buffer[out];
out = (out + 1) % BUFFER_SIZE;
counter--;
I* consume the item in nextConsumed *I
}
```

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

As an illustration, suppose that the value of the variable counter is currently 5 and that the producer and consumer processes execute the statements "counter++" and "counter--" concurrently. Following the execution of these two statements, the value of the variable counter may be 4, 5, or 6! The only correct result, though, is counter == 5, which is generated correctly if the producer and consumer execute separately.

We can show that the value of counter may be incorrect as follows. Note that the statement" counter++" may be implemented in machine language (on a typical machine) as

```
register1 = counter
register1 = register1 + 1
counter= register1
```

where register1 is one of the local CPU registers. Similarly, the statement register2"counter--" is implemented as follows:

```
register2 = counter
register2 = register2 - 1
counter= register2
```

where again register2 is on eo£ the local CPU registers. Even though register1 and register2 may be the same physical register (an accumulator, say), remember that the contents of this register will be saved and restored by the interrupt handler.

The concurrent execution of "counter++" and "counter--" is equivalent to a sequential execution in which the lower-level statements presented previously are interleaved in some arbitrary order (but the order within each high-level statement is preserved). One such interleaving is

```
To: producer execute register1 =counter {register1 = 5}
T1: producer execute register1 = register1 + 1 {register1 = 6}
T2: consumer execute register2 = counter {register2 = 5}
T3: consumer execute register2 = register2 - 1 {register2 = 4}
T4: producer execute counter= register1 {counter = 6}
Ts: consumer execute counter = register2 {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".

We would arrive at this incorrect state because we allowed both processes to manipulate the variable counter concurrently. 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.

Situations such as the one just described occur frequently in operating systems as different parts of the system manipulate resources. Furthermore, with the growth of multicore systems, there is an increased emphasis on developing multithreaded applications wherein several threads-which are quite possibly sharing data-are rmming in parallel on different processing cores. Clearly, we want any changes that result from such activities not to interfere with one another. Because of the importance of this issue, a major portion of this chapter is concerned with process synchronization and coordination amongst cooperating processes.

5.2 Critical Section Problem

Consider a system consisting of n processes {Po, P1 , ... , P11 _ 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. That is, no two processes are executing in their critical sections at the same time. The critical-section problem is to design a protocol that the processes can use to cooperate. Each process must request permission to enter its critical section. The section of code implementing this request is the entry section. The critical section may be followed by an exit section. The remaining code is the remainder section. The general structure of a typical process Pi is shown in Figure. The entry section and exit section are enclosed in boxes to highlight these important segments of code.

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

Figure 5.2 : Structure of a process

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 deciding 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.

We assume that each process is executing at a nonzero speed. However, we can make no assumption concerning the relative speed of the n processes. At a given point in time, many kernel-mode processes may be active in the operating system. As a result, the code implementing an operating system (kernel code) is subject to several possible race conditions. Consider as an example a kernel data structure that maintains a list of all open files in the system. This list must be modified when a new file is opened or closed (adding the file to the list or removing it from the list). If two processes were to open files simultaneously, the separate updates to this list could result in a race condition. Other kernel data structures that are prone to possible race conditions include structures for maintaining memory allocation, for maintaining process lists, and for interrupt handling. It is up to kernel developers to ensure that the operating system is free from such race conditions.

Two general approaches are used to handle critical sections in operating systems: (1) preemptive kernels and (2) non-preemptive kernels. A preemptive kernel allows a process to be preempted while it is running in kernel mode.

A non-preemptive kernel does not allow a process running in kernel mode to be preempted; a kernel-mode process will run until it exits kernel mode, blocks, or voluntarily yields control of the CPU. Obviously, a non-preemptive 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. Preemptive kernels are especially difficult to design for SMP architectures, since in these environments it is possible for two kernel-mode processes to run simultaneously on different processors.

Why, then, would anyone favor a preemptive kernel over a non-preemptive one? A preemptive kernel is more suitable for real-time programming, as it will allow a real-time process to preempt a process currently running in the kernel. Furthermore, a preemptive kernel may be more responsive, since there is less risk that a kernel-mode process will run for an arbitrarily long period before relinquishing the processor to waiting processes. Of course, this effect can be minimized by designing kernel code that does not behave in this way. Later in this chapter, we explore how various operating systems manage preemption within the kernel.

5.3 Semaphores

The hardware-based solutions to the critical-section problem presented are complicated for application programmers to use. To overcome this difficulty, we can use a synchronization tool called a semaphore. A semaphore S is an integer variable that, apart from initialization, is accessed only through two standard atomic operations: wait () and signal ().

The wait () operation was originally termed P (from the Dutch proberen, "to test"); signal() was originally called V (from verhogen, "to increment"). The definition of wait () is as follows:

```
wait(S) {
while S <= 0
II no-op
S--;
}
```

The definition of signal() is as follows:

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

All 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. In addition, in the case of wait (S), the testing of the integer value of S (S :S 0), as well as its possible modification (S--), must be executed without interruption. We shall see how these operations can be implemented let us see how semaphores can be used.

5.3.1 The Usage

Operating systems often distinguish between counting and binary semaphores. The value of a counting semaphore can range over an unrestricted domain. The value of a binary semaphore can range only between 0 and 1. On some systems, binary semaphores are lmown as mutex locks, as they are locks that provide mutual exclusion.

We can use binary semaphores to deal with the critical-section problem £or mlJltiple processes. Then processes share a semaphore, mutex, initialized to 1. Each process Pi is organized. Counting semaphores can be used to control access to a given resource consisting of a finite number o£ 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 (thereby 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.

We can also use semaphores to solve various synchronization problems. For example, consider two concurrently numing processes: P1 with a statement 51 and P2 with a statement 52 . Suppose we require that 52 be executed only after 51 has completed. We can implement this scheme readily by letting P1 and P2 share a common semaphore synch, initialized to 0, and by inserting the statements

```
51;
signal(synch) ;
```

in process P1 and the statements

```
wait(synch);
52;
```

in process P2. Because synch is initialized to 0, P2 will execute 52 only after P1 has invoked signal (synch), which is after statement 51 has been executed.

5.3.2 Implementation

The main disadvantage of the semaphore definition given here is thatit requires 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 a real multiprogramming system,

```
do {
    wait (mutex) ;
    II critical section
    signal(mutex);
    II remainder section
} while (TRUE);
```

where a single CPU is shared among ncany processes. Busy waiting wastes CPU cycles that some other process might be able to use productively. This type of semaphore is also called a because the process "spins" while waiting for the lock. (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. When a process executes the wait () operation and finds that the semaphore value is not positive, it must wait. However, 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.)

To implement semaphores under this definition, we define a semaphore as a "C" struct:

```
typedef struct {
    int value;
    struct process *list;
} semaphore;
```

Each semaphore has an integer value and a list of processes list. 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 fron< 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 that in this implementation, semaphore values may be negative, although semaphore values are never negative under the classical definition of semaphores with busy waiting. If a semaphore value is negative, its magnitude is the number of processes waiting on that semaphore. This is fact results from switching the order of the decrement and the test in the implementation of the wait () operation.

The list of waiting processes can be easily implemented by a link field in each process control block (PCB). Each semaphore contains an integer value and a pointer to a list of PCBs. One way to add and remove processes from the list so as to ensure bounded waiting is to use a FIFO queue, where the semaphore contains both head and tail pointers to the queue. In general, however the list can use any queuing strategy. Correct usage of semaphores does not depend on a particular queuing strategy for the semaphore lists.

It is critical that semaphores be executed atomically. We must guarantee that no two processes can execute wait() and signal() operations on the same semaphore at the same time. This is a critical-section problem; and in a single-processor environment (that is, where only one CPU exists), we can solve it by simply inhibiting interrupts during the time the wait() and signal() operations are executing. This scheme works in a single-processor environment because, once interrupts are inhibited, instructions from different processes cannot be interleaved. Only the currently running process executes until interrupts are re-enabled and the scheduler can regain control.

In a multiprocessor environment, interrupts must be disabled on every processor; otherwise, instructions from different processes (running on different processors) may be interleaved in some arbitrary way. Disabling interrupts on every processor can be a difficult task and furthermore can seriously diminish performance. Therefore, SMP systems must provide alternative locking techniques-such as spinlocks-to ensure that wait() and signal() are performed atomically.

It is important to admit that we have not completely eliminated busy waiting with this definition of the wait () and signal () operations. Rather, we have moved busy waiting from the entry section to the critical sections of application programs. Furthermore, we have limited busy waiting to the critical sections of the wait () and signal () opera times, and these sections are short (if properly coded, they should be no more than about ten instructions). Thus, the critical section is almost never occupied, and busy waiting occurs rarely, and then for only a short time. An entirely different situation exists with application programs whose critical sections may be long (minutes or even hours) or may almost always be occupied. In such case of busy waiting is extremely inefficient.

5.4 Shared Memory Multiprocessor

Processes that share memory can exchange data by writing and reading shared variables. As an example, consider two processes p and q that share some variable s. Then p can communicate information to q by writing new data in s, which q can then read.

The above discussion raises an important question. How does q know when p writes new information into s? In some cases, q does not need to know. For instance, if it is a load balancing program that simply looks at the current load in p's machine stored in s. When it does need to know, it could poll, but polling puts undue burden on the cpu. Another possibility is that it could get a software interrupt, which we discuss below. A familiar alternative is to use semaphores or conditions. Process q could block till p changes s and sends a signal that unblocks q. However, these solutions would not allow q to (automatically) block if s cannot hold all the data it wants to write. (The programmer could manually implement a bounded buffer using semaphores.) Moreover, conventional shared memory is accessed by processes on a single machine, so it cannot be used for communicating information among remote processes. Recently, there has been a lot of work in distributed shared memory over LANs, which you will study in 203/243, which tends to be implemented using inter-process communication. However, even if we could implement shared memory directly over WANs (without message passing), it is not an ideal abstraction for all kinds of IPC. In particular, it is not the best abstraction for sending requests to servers, which requires coding of these requests as data structures (unless these data structures are encapsulated in monitors, whom we shall study later). As we shall see later, message passing (in particular, RPC) is more appropriate for supporting client-server interaction.

5.4.1 Software Interrupt

We can introduce a service call that lets one process cause a software interrupt in another: Interrupt(process id, interrupt number) and another that allows a process to associate a handler with an interrupt: Handle(interrupt number, handler)

Software interrupts allow only one bit information to be communicate - that an event associated with the interrupt number has occurred. They are typically used by an operating system to inform a process about the following events:

The user typed the "attention key". An alarm scheduled by the process has expired. Some limit, such as file size or virtual time, has been exceeded. It is important to distinguish among interrupts, traps, software interrupts, and exceptions. In all cases, an event is processed asynchronously by some handler procedure. Interrupt and trap numbers are defined by the hardware which is also responsible for calling the procedure in the kernel space. An interrupt handler is called in response to a signal from another device while a trap handler is called in response to an instruction executed within the cpu.

Software interrupt and exception handlers are called in user space. A software interrupt handler is called in response to the invocation of a system call. Software interrupt numbers are defined by the operating system. Exceptions are defined and processed by the programming language. An exception raised in some block, b, of some process p, can be caught by a handler in the same block, or a block/procedure (in p) along static/dynamic links from b, or by a process q that (directly or indirectly) forked p. The raiser of an exception does not identify which process should handle it, so exceptions are not IPC mechanisms.

The notion of software interrupts is somewhat confused in some environments such as the PC, where traps to kernel-provided I/O routines are called software interrupts. There is a special instruction on the PC called INT which is used to invoke these traps. For instance, the instruction int 16H executes the BIOS interrupt routine for processing the current character received from the keyboard. (It is executed by the interrupt handler of the Xinu kernel to ask the PC BIOS handler to fetch the character from the keyboard.) The term interrupt is used because these routines are called usually by hardware interrupt routines. We are using the term software interrupts for what Unix calls signals, which are not to be confused with semaphores, though you invoke the signal operation on both!

5.5 Classical Problem of Synchronization

Following are some of the classical problem faced while process synchronization in system where cooperating process are present.

5.5.1 Bounded Buffer Problem

- This problem is generalised in term of the producer-consumer problem.

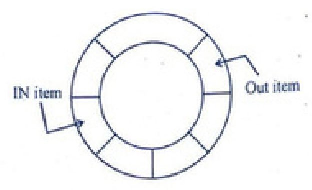


Figure 5.1.1 : Producer-Consumer problem

- There are two groups of processes, producers & consumers. Each producer deposits a data items into in position and advance the pointer in & each consumer retrieves the data item in position out & advances the pointer out.
- A producer cannot deposit its data if the buffer is full. Similarly, a consumer cannot retrive any data if the buffer is empty. On the other hand, if the buffer is not full, a producer can deposit its data.
- Solution to this problem is, creating two counling semaphose "fill" & "empty" to keep track of the current number of full & empty buffers respectively.

5.5.2 The Readers Writers Problem

Reader- writers problem are examples of a common computing problem in concurrency. The two problems deal with situations in which many threads must access the same shared memory at one some reading & some writing. With the natural constraint that no process may access the share for reading or writing while another process is in the act writing to it.

A reader-writer lock is a data structure that solves one or more of the readers writer problems.

5.5.3 Dining Philosopher Problem

The dining philosopher's problem involves the the allocation of limited resources from a group of processes in a deadlock free & starvation free manner.

There are five philosophers setting around a table, in which there are 5 cop sticks kept besede them & a bowl of rice in the centre, when a philosopher want to eat, he uses two chopsticks one from their left & one from their right. When a philosopher want to think, he keeps down both chopsticks at their original place. This problem is referred to as dining philosopher problem.

One of the solution using the semaphores.

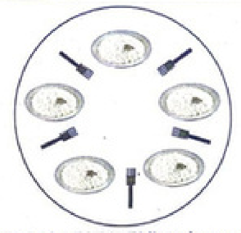


Figure 5.5.3 : Dining Philosopher problem

5.5.4 Sleeping barber problem

- It is a classic interprocess communication & synchronization problem between multiple operating system processes.
- The problem is analogous to that of keeping a barber working when there are customers, resting when there are none & doing so in an orderly manner.
- Not implementing a proper solution can lead to the usual inter process communication problems of starvation & deadlock.
- For example, the barber could end up waiting on a customer & a customer waiting on the barber, resulting in deadlock. Alternatively, customers may not decide to approach the barber in an orderly manner, leading to process starvation as some customers never get the chance for a haircut even though they have been waiting.
- This problem involves only one barber, & it is therefore also called the single sleeping barber problem. A multiple sleeping barbers problem is similar in the nature of implementation but has the additional complexity of coordinating several barber among the waiting customers.
- Here is one implementation of the suggested solution is using semaphore.

5.6 Inter-Process Communication

Processes executing concurrently in the operating system may be either independent processes or cooperating processes. A process is independent if it cannot affect or be affected by the other processes executing in the system. Any process that does not share data with any other process is independent. A process is cooperating if it can affect or be affected by the other processes executing in the system. Clearly, any process that shares data with other processes is a cooperating process.

There are several reasons for providing an environment that allows process cooperation:

- Information sharing. Since several users may be interested in the same piece of information (for instance, a shared file), we must provide an environment to allow concurrent access to such information.
- Computation speedup. If we want a particular task to run faster, we must break it into subtasks, each of which will be executing in parallel with the others. Notice that such a speedup can be achieved only if the computer has multiple processing elements (such as CPUs or I/O channels).
- Modularity. We may want to construct the system in a modular fashion, dividing the system functions into separate processes or threads.
- Convenience. Even an individual user may work on many tasks at the same time. For instance, a user may be editing, printing, and compiling in parallel.

Cooperating processes require an inter-process communication (IPC) mechanism that will allow them to exchange data and information. There are two fundamental models of inter-process communication: (1) shared memory and (2) message passing. In the shared-memory model, a region of memory that is shared by cooperating processes is established. Processes can then exchange information by reading and writing data to the shared region. In the message passing

model, communication takes place by means of messages exchanged between the cooperating processes.

Both of the models just discussed are common in operating systems, and many systems implement both. Message passing is useful for exchanging smaller amounts of data, because no conflicts need be avoided. Message passing is also easier to implement than is shared memory for inter-computer communication. Shared memory allows maximum speed and convenience of communication. Shared memory is faster than message passing, as message passing systems are typically implemented using system calls and thus require the more time-consuming task of kernel intervention. In contrast, in shared memory systems, system calls are required only to establish shared-memory regions. Once shared memory is established, all accesses are treated as routine memory accesses, and no assistance from the kernel is required. In the remainder of this section, we explore each of these IPC models in more detail.

5.7 Message Passing

In last Section, we showed how cooperating processes can communicate in a shared-memory environment. The scheme requires that these processes share a region of memory and that the code for accessing and manipulating the shared memory be written explicitly by the application programmer. Another way to achieve the same effect is for the operating system to provide the means for cooperating processes to communicate with each other via a message-passing facility. Irt Section 3.4.1, we showed how cooperating processes can communicate in a shared-memory environment. The scheme requires that these processes share a region of memory and that the code for accessing and manipulating the shared memory be written explicitly by the application

programmer. Another way to achieve the same effect is for the operating system to provide the means for cooperating processes to communicate with each other via a message-passing facility.

Message passing provides a mechanism to allow processes to communicate and to synchronize their actions without sharing the same address space and is particularly useful in a distributed environment, where the communicating processes may reside on different computers connected by a network. For example, a chat program used on the World Wide Web could be designed so that chat participants communicate with one another by exchanging messages.

The most popular form of inter-process communication involves message passing. Processes communicate with each other by exchanging messages. A process may send information to a port, from which another process may receive information. The sending and receiving processes can be on the same or different computers connected via a communication medium.

One reason for the popularity of message passing is its ability to support client-server interaction. A server is a process that offers a set of services to client processes. These services are invoked in response to messages from the clients and results are returned in messages to the client. Thus a process may act as a web search server by accepting messages that ask it to search the web for a string. In this course we shall be particularly interested in servers that offer operating system services. With such servers, part of the operating system functionality can be transferred from the kernel to utility processes.

For instance file management can be handled by a file server, which offers services such as open, read, write, and seek. Similarly, terminal management can also be handled by a server that offers services such as getchar and putchar.

A message-passing facility provides at least two operations: send(message) and receive(message). Messages sent by a process can be of either fixed or variable size. If only fixed-sized messages can be sent, the system-level implementation is straightforward. This restriction, however, makes the task item next Consumed;

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

of programming more difficult. Conversely, variable-sized messages require a lnore complex system-level implementation, but the programming task becomes simpler. This is a common kind of trade-off seen throughout operating system design.

If processes P and Q want to communicate, they must send messages to and receive messages from each other; a communication link must exist between them. This link can be implemented in a variety of ways.

5.8 Mailboxes

With indirect communication, the messages are sent to and received from mailboxes, or ports. A mailbox can be viewed abstractly as an object into which messages can be placed by processes and from which messages can be removed. Each mailbox has a unique identification. For example, POSIX message queues use an integer value to identify a mailbox. In this scheme, a process can communicate with some other process via a number of different mailboxes.

Two processes can communicate only if the processes have a shared mailbox, however. The send() and receive 0 primitives are defined as follows:

- Send (A, message) -Send a message to mailbox A.
- Receive (A, message)-Receive a message from mailbox A. In this scheme, a communication link has the following properties:
- A link is established between a pair of processes only if both members of the pair have a shared mailbox.
- A link may be associated with more than two processes.
- Between each pair of communicating processes, there may be a number of different links, with each link corresponding to one mailbox.

Now suppose that processes P1, P2, and P3 all share mailbox A. Process P1 sends a message to A, while both P2 and P3 execute a receive 0 from A. Which process will receive the message sent by P1? The answer depends on which of the following methods we choose:

- Allow a link to be associated with two processes at most.
- Allow at most one process at a time to execute a receive 0 operation.
- Allow the system to select arbitrarily which process will receive the message (that is, either P2 or P3, but not both, will receive the message). The system also may define an algorithm for selecting which process will receive the message (that is, round robin, where processes take turns receiving messages). The system may identify the receiver to the sender.

A mailbox may be owned either by a process or by the operating system. If the mailbox is wned by a process (that is, the mailbox is part of the address space of the process), then we stinguish between the owner (which can only receive messages through this mailbox) and the ser (which can only send messages to the mailbox). Since each mailbox has a unique owner, there can be no confusion about which process should receive a message sent to this mailbox. When a process that owns a mailbox terminates, the mailbox disappears. Any process that subsequently sends a message to this mailbox must be notified that the mailbox no longer exists.

In contrast, a mailbox that is owned by the operating system has an existence of its own. It is independent and is not attached to any particular process. The operating system then must provide a mechanism that allows a process to do the following:

- Create a new mailbox.
- Send and receive messages through the mailbox.

- Delete a mailbox.

The process that creates a new mailbox is that mailbox's owner by default. Initially, the owner is the only process that can receive messages through this mailbox. However, the ownership and receiving privilege may be passed to other processes through appropriate system calls. Of course, this provision could result in multiple receivers for each mailbox.

Exercise

Part I (Very Short Answer)

1. What is Coordinating process?
2. What is a software interrupt?
3. Write about critical section.
4. Write about Shared memory.

Part II (Short Answer)

5. .Write in brief about mail boxes.
6. Compare different techniques of Inter-process Communications.
7. Write about interactive and coordinating process in brief.

Part III (Long Answer)

8. What is critical section? What can be the Solutions of critical section problem? Explain.
9. Explain Semaphores with the help of example.
10. Describe the problem of synchronization ?

■■■

Chapter

6 Deadlock

A deadlock is a situation in which two computer programs sharing the same resource are effectively preventing each other from accessing the resource, resulting in both programs ceasing to function.

Operating system is a resource allocator. There are many resources that can be allocated to only one process at a time, and we have seen many operating system features that allow this.

Sometimes a process has to reserve more than one resource. For example, a process which copies files from one tape to another generally requires two tape drives. A process which deals with databases may need to lock multiple records in a database.

Is it a state where two ore more operations are waiting for each other, say a computing action 'A' is waiting for action 'B' to complete, while action 'B' can only execute when 'A' is completed. Such a situation would be called a deadlock. In operating systems, a deadlock situation is arrived when computer resources required for complete of a computing task are held by another task that is waiting to execute. The system thus goes into an indefinite loop resulting into a deadlock.

The deadlock in operating system seems to be a common issue in multiprocessor systems, parallel and distributed computing setups.

Four (4) necessary conditions that must hold simultaneously for there to be a deadlock.

1. **Mutual Exclusion Condition :** Only one process at a time claims exclusive control of the resource. If another process requests that resource, the requesting process must be delayed until the resource has been released.

2. **Hold and Wait Condition :** A process that is holding a resource already allocated to it while waiting for additional resource that are currently being held by other processes.

3. **No-Preemptive Condition :** Resources cannot be removed from the processes are used to completion or released by the process holding it.

4. **Circular Wait Condition :** Each process in the list is waiting for a resource held by the next process in the list.

6.1 System Model

- A system can be modeled as a collection of limited resources, which can be partitioned into different categories, to be allocated to a number of processes, each having different needs.
- Resource categories may include memory, printers, CPUs, open files, tape drives, CD-ROMS, etc.

- By definition, all the resources within a category are equivalent, and a request of this category can be equally satisfied by any one of the resources in that category. If this is not the case (i.e. if there is some difference between the resources within a category), then that category needs to be further divided into separate categories. For example, "printers" may need to be separated into "laser printers" and "color inkjet printers".
- Some categories may have a single resource.
- In normal operation a process must request a resource before using it, and release it when it is done, in the following sequence:

1. Request - If the request cannot be immediately granted, then the process must wait until the resource(s) it needs become available. For example the system calls open( ), malloc( ), new( ), and request( ).

2. Use - The process uses the resource, e.g. prints to the printer or reads from the file.

3. Release - The process release the resource. so that it becomes available for other processes. For example, close( ), free( ), delete( ), and release( ).

Necessary Conditions

In order for deadlock to occur, four conditions must be true.

Mutual Exclusion - Each resource is either currently allocated to exactly one process or it is available. (Two processes cannot simultaneously control the same resource or be in their critical section).

Hold and Wait - A process must be simultaneously holding at least one resource and waiting for at least one resource that is currently being held by some other process.

No preemption - Once a process is holding a resource (i.e. once its request has been granted), then that resource cannot be taken away from that process until the process voluntarily releases it.

Circular Wait - Each process is waiting to obtain a resource which is held by another process.A set of processes { P0, P1, P2, . . ., PN } must exist such that every P[i] is waiting for P[(i + 1) % (N + 1)].

6.2 Resource-Allocation Graph

Deadlocks can be understood more clearly through the use f Resource-Allocation Graphs, having the following properties:

1. A set of directed arcs from Pi to Rj, indicating that process Pi has requested Rj, and is currently waiting for that resource to become available, that is Request Edges.
2. A set of directed arcs from Rj to Pi indicating that resource Rj has been allocated to process Pi, and that Pi is currently holding resource Rj,that is Assignment Edges.
3. A request edge can be converted into an assignment edge by reversing the direction of the arc when the request is granted. (However note also that request edges point to the category box, whereas assignment edges emanate from a particular instance dot within

the box.)

4. A set of resource in the system R1, R2, R3, . . ., Rn , which appear as square nodes on the graph. Dots inside the resource nodes indicate specific instances of the resource.

5. A set of processes, { P1, P2, P3, . . ., Pn }

For example:

Process State:

- P1 is holding an instance of R2 and waiting for an instance of R1.

- P2 is holding an instance of R1 and R2, and is waiting for an instance of R3.

- P3 is holding an instance of R3.

● The graph does not contain any cycles. What does that mean?

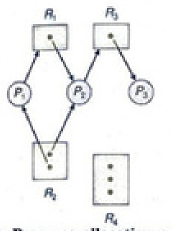


Figure 6.2 (a): Resource allocation graph

● Two minimal cycles exist in the system:
- P1→R1→P2→R3→P3→R2→P1
- P2→R3→P3→R2→P2

● Processes P1, P2, P3 are deadlocked.
- P2 is waiting for R3, which is held by P3.
- P3 is waiting for P1 or P2 to release R2.
- P1 is waiting for P2 to release R1.

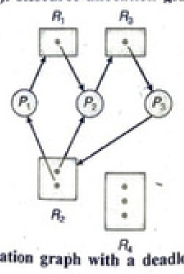


Figure 6.2 (b) : Resource allocation graph with a deadlock

We have a cycle:

P1→R1→P3→R2→P1

● There is no deadlock.

● P4 may release its instance of R2 and that resource can then be allocated to P3 breaking the cycle.

● If graph contains no cycles ?no deadlock.

● If graph contains a cycle ?

- if only one instance per resource type, then deadlock.

- if several instances per resource type, possibility of deadlock.

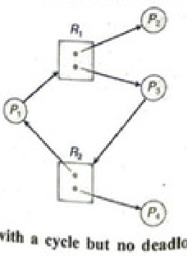


Figure 6.2 (c) : Resource allocation graph with a cycle but no deadlock

6.3 Deadlock Prevention

By ensuring that at least one of these conditions cannot hold, we can prevent the occurrence of a deadlock.

There must be following four conditions that must hold simultaneously for a deadlocks to occur:

1. Mutual exclusion
2. Hold and Wait
3. No preemption
4. Circular wait

1. Mutual exclusion

- Not required for sharable resources; must hold for non-sharable resources.
- For example, a printer cannot be simultaneously shared by several processes.
- A process never needs to wait for a sharable resource.

2. Hold and Wait

- Must guarantee that whenever a process requests a resource, it does not hold any other resources.
- One protocol requires each process to request and be allocated all its resources before it begins execution,
- Or another protocol allows a process to request resources only when the process has none. So, before it can request any additional resources, it must release all the resources that it is currently allocated.
- Example to illustrate the difference between the two protocols: A process copies data from a tape drive to a disk file, sorts the disk file, and then prints the results to a printer.

$$\begin{tikzcd} \boxed{\text{Tape}} \arrow[r] & \boxed{\text{Disk}} \arrow[r] & \boxed{\text{Printer}} \end{tikzcd}$$

- Protocol 1: request all tape, disk, printer and hold for entire execution; note printer will be idle for a long time.

- Protocol 2: request tape and disk only after copying release both then request disk and printer after printing release both.

Two main disadvantages to these protocols:

1. Low resource utilization: since many of the resources may be allocated but unused for a long time.

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

3. No preemption : To ensure that this condition does not hold, we can use the following protocol:

- If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released.
- Preempted resources are added to the list of resources for which the process is waiting.
- Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.

4. Circular wait

- To ensure that the circular-wait condition never holds is to determine a total ordering of all resource types, and to require that each process requests resources in an increasing order of enumeration.
- Example: Let R={R1, R2, ..., Rm} be the set of resource types.Assign to each resource type a unique integer number to compare two resources and to determine whether one proceeds another in ordering.
- For example: If the set of resource types R includes tape drives, disk drives, and printers, then a function F might be defined as follows:

  F(tape drive)= 1; F(disk drive)= 5; F(Printer)= 12.
- A protocol to prevent deadlocks: Each process can request resources only in an increasing order of enumeration.
- A process can initially request any number of instances of a resource type Ri.

1. That process can request instances of resource type Rj if and only if F(Rj)>F(Ri). From the previous example; a process wants to use the tape drive and printer at the same time, must first request the tape drive and then the printer.

2. When a process requests an instance of resource type Rj, it has released any resources Ri such that F(Ri)>=F(Rj).

- By applying 1 and 2 then the circular-wait condition cannot hold.

6.4 Deadlock Avoidance

Deadlock-prevention algorithms, as discussed in previous, prevent deadlocks by restraining how requests can be made. The restraints ensure that at least one of the necessary conditions for deadlock cannot occur and, hence, that deadlocks cannot hold. Possible side effects of preventing deadlocks by this method, however, are 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. For example, in a system with one tape drive and one printer, the system might need to know that process P will request first the tape drive and then the printer before releasing both resources, whereas process Q will request first the printer and then the tape drive. With this knowledge of the complete sequence of requests and releases for each process, the system can decide for each request whether or not the process should wait in order to avoid

a possible future deadlock. Each request requires that in making this decision the system consider the resources currently available, the resources currently allocated to each process, and the future requests and releases of each process.

The various algorithms that use this approach differ in the amount and type of information required. The simplest and most useful model requires that each process declare the maximum number of resources of each type that it may need. Given this a priori information, it is possible to construct an algorithm that ensures that the system will never enter a deadlocked state. Such an algorithm defines the deadlock-avoidance approach. A deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that a circular wait condition can never exist. The resource-allocation state is defined by the number of available and allocated resources and the maximum demands of the processes. In the following sections, we explore two deadlock-avoidance algorithms.

6.4.1 The 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. A sequence of processes <P1, P2, ... , Pn> is a safe sequence for the current allocation state if, for each Pi, the resource requests that Pi can still make can be satisfied by the currently available resources plus the resources held by all Pj, with j < i. In this situation, if the resources that Pi needs are not immediately available, then Pi can wait until all Pj have finished. When they have finished, Pi can obtain all of its needed resources, complete its designated task, return its allocated resources, and terminate. When Pi terminates, Pi+l can obtain its needed resources, and so on. If no such sequence exists, then the system state is said to be unsafe.

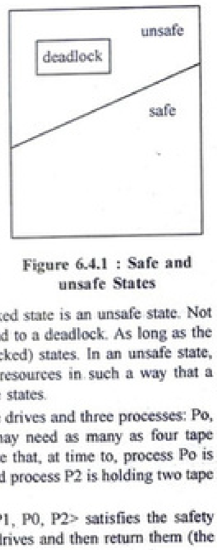


Figure 6.4.1 : Safe and unsafe States

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

To illustrate, we consider a system with twelve magnetic tape drives and three processes: Po, P1, and P2. Process Po requires ten tape drives, process P1 may need as many as four tape drives, and process P2 may need up to nine tape drives. Suppose that, at time to, process Po is holding five tape drives, process P1 is holding two tape drives, and process P2 is holding two tape drives (Thus, there are three free tape drives.)

At time t0, the system is in a safe state. The sequence <P1, P0, P2> satisfies the safety condition. Process P1 can immediately be allocated all its tape drives and then return them (the system will then have five available tape drives); then process Po can get all its tape drives and return them (the system will then have ten available tape drives); and finally process P2 can get all its tape drives and return them (the system will then have all twelve tape drives available). A system can go from a safe state to an unsafe state. Suppose that, at time t1, process P2 requests and is allocated one more tape drive. The system is no longer in a safe state. At this point, only process P1 can be allocated all its tape drives. When it returns them, the system will have only four available tape drives. Since process Po is allocated five tape drives but has a maximum of ten, it may request five more tape drives. If it does so, it will have to wait, because they are unavailable. Similarly, process P2 may request six additional tape drives and have to wait, resulting in a deadlock. Our mistake was in granting the request from process P2 for one more tape drive. If we had made P2 wait until either of the other processes had finished and released its resources, then we could have avoided the deadlock.

6.4.2 The Bankers Algorithm

The resource-allocation-graph algorithm is not applicable to a resource allocation system with multiple instances of each resource type. The deadlock avoidance algorithm that we describe next is applicable to such a system but is less efficient than the resource-allocation graph scheme. This algorithm is commonly known as the banker's algorithm. The name was chosen because the algorithm. could be used in a banking system to ensure that the bank never allocated its available cash in such a way that it could no longer satisfy the needs of all its customers. When a new process enters the system, it must declare the maximum number of instances of each resource type that it may need. This nun1.ber 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.

Several data structures must be maintained to implement the banker's algorithm. These data structures encode the state of the resource-allocation system. We need the following data structures, where n is the number of processes in the system and m is the number of resource types:

- Available: A vector of length m indicates the number of available resources of each type. If Available[j] equals k, then k instances of resource type Ri are available.
- Max. An n x m matrix defines the maximum demand of each process.If Max[i] [j] equals k, then process P; may request at most k instances of resource type Ri.
- Allocation. An 11 x m matrix defines the number of resources of each type currently allocated to each process. If Allocation[i][j] equals lc, then process P; is currently allocated lc instances of resource type Rj.
- Need. An n x m matrix indicates the remaining resource need of each process. If Need[i][j] equals k, then process P; may need k more instances of resource type Ri to complete its task. Note that Need[i][j] equals Max[i][j] - Allocation [i][j].

These data structures vary over time in both size and value.

To simplify the presentation of the banker's algorithm, we next establish some notation. Let X andY be vectors of length 11. We say that X::= Y if and only if X[i] ::= Y[i] for all i = 1, 2, ... , n. For example, if X = (1,7,3,2) and Y = (0,3,2,1), then Y ::=X. In addition, Y < X if Y ::=X andY# X.

We can treat each row in the matrices Allocation and Need as vectors and refer to them as Allocation; and Need;. The vector Allocation; specifies the resources currently allocated to process P;; the vector Need; specifies the additional resources that process P; may still request to complete its task.

6.5 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 may provide:

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

In the following discussion, we elaborate on these two requirements as they pertain to systems with only a single instance of each resource type, as well as to systems with several instances of each resource type. At this point, however, we note that a detection-and-recovery scheme requires overhead that includes not only the run-time costs of maintaining the necessary information and executing the detection algorithm but also the potential losses inherent in recovering from a deadlock.

6.5.1 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. We obtain this graph from the resource-allocation graph by removing the resource nodes and collapsing the appropriate edges. More precisely, an edge from Pi to Pi in a wait-for graph implies that process Pz is waiting for process P1 to release a resource that P; needs. An edge Pz --+ Pi exists iil a wait-for graph if and only if the corresponding resource allocation graph contains two edges Pz --+ Rq and Rq --+ Pi for some resource Rq.

As before, 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 n2 operations, where n is the number of vertices in the graph.

6.5.2 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 that is applicable to such a system. The algorithm employs several time-varying data structures that are similar to those used in the banker's algorithm:

- Available: A vector of length nz indicates the number of available resources of each type.
- Allocation. Ann x nz matrix defines the number of resources of each type currently allocated to each process.
- Request. An n x m matrix indicates the current request of each process. If Request[i][j] equals k, then process P; is requesting k more instances of resource type Rj.

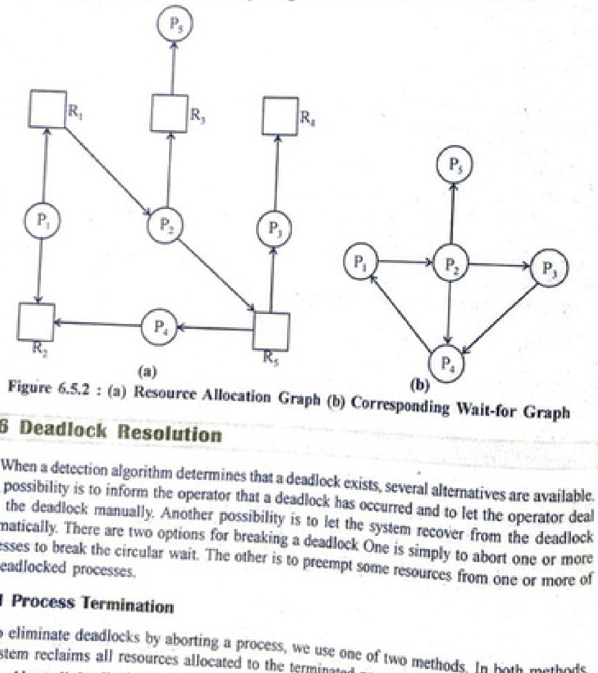


Figure 6.5.2 : (a) Resource Allocation Graph (b) Corresponding Wait-for Graph

6.6 Deadlock Resolution

When a detection algorithm determines that a deadlock exists, several alternatives are available. One possibility is to inform the operator that a deadlock has occurred and to let the operator deal with the deadlock manually. Another possibility is to let the system recover from the deadlock automatically. There are two options for breaking a deadlock One is simply to abort one or more processes to break the circular wait. The other is to preempt some resources from one or more of the deadlocked processes.

6.6.1 Process Termination

To eliminate deadlocks by aborting a process, we use one of two methods. In both methods, the system reclaims all resources allocated to the terminated processes.

- Abort all deadlocked processes. This method clearly will break the deadlock cycle, but at great expense; the deadlocked processes may have computed for a long time, and the results of these partial computations must be discarded and probably will have to be recomputed later.

- Abort one process at a time until the deadlock cycle is eliminated. This method incurs considerable overhead, since after each process is aborted, a deadlock-detection algorithm must be invoked to determine whether any processes are still deadlocked.

Aborting a process may not be easy. If the process was in the midst of updating a file, terminating it will leave that file in an incorrect state. Similarly, if the process was in the midst of printing data on a printer, the system must reset the printer to a correct state before printing the next job. If the partial termination method is used, then we must determine which deadlocked process (or processes) should be terminated. This determination is a policy decision, similar to CPU-scheduling decisions. The question is basically an economic one; we should abort those processes whose termination will incur the minimum cost. Unfortunately, the term minimum cost is not a precise one.

Many factors may affect which process is chosen, including:

1. What the priority of the process is
2. How long the process has computed and how much longer the process will compute before completing its designated task
3. How many and what types of resources the process has used (for example, whether the resources are simple to preempt)
4. How many more resources the process needs in order to complete
5. How many processes will need to be terminated Whether the process is interactive or batch

6.6.2 Resource Preemption

To eliminate deadlocks using resource preemption, we successively pre-empt some resources from processes and give these resources to other processes 1-m til the deadlock cycle is broken. If preemption is required to deal with deadlocks, then three issues need to be addressed:

1) Selecting a victim. Which resources and which processes are to be preempted? As in process termil<ation, we must determine the order of preemption to minimize cost. Cost factors may include such parameters as the number of resources a deadlocked process is holding and the amount of time the process has thus far consumed during its execution.

2) Rollback. If we preempt a resource from a process, what should be done with that process? Clearly, it cannot contil<ue with its normal execution; it is missing some needed resource. We must roll back the process to some safe state and restart it from that state. Since, in general, it is difficult to determine what a safe state is, the simplest solution is a total rollback: abort the process and then restart it. Although it is more effective to roll back the process only as far as necessary to break the deadlock, this method requires the system to keep more information about the state of all running processes.

3) Starvation. How do we ensure that starvation will not occur? That is, how can we guarantee that resources will not always be preempted from the same process?

In a system where victim selection is based primarily on cost factors, it may happen that the same process is always picked as a victim. As a result, this process never completes its designated task, a starvation situation that must be dealt with in any practical system. Clearly, we must ensure that a process can be picked as a victim" only a (small) finite number of times. The most common solution is to include the number of rollbacks in the cost factor.

Exercise

Part I (Very Short Answer)

1. What is situation of deadlock?
2. What is a mutual exclusion?
3. What is deadlock avoidance?
4. What is deadlock detection?
5. What do you mean by roll back.

Part II (Short Answer)

6. .Write in brief the necessary conditions of deadlock.
7. Write in brief about deadlock prevention.
8. Write in brief about process

Part-III (Long Answer)

9. Explain Resource allocation graphs with example.
10. Explain deadlock prevention in all four conditions.