॥ सरस्वती नः सुभगा मयस्करत् ॥

Uttar Pradesh Rajarshi Tandon
Open University

Master of Computer Applications

DCECS - 106

Operating System

Uttar Pradesh Rajarshi Tandon
Open University

Master of Computer Applications

DCECS - 106

Operating System

BLOCK

1

Block-1 An Overview and Process Management

UPRTOU, Prayagraj-2022
DCECS -106 Operating System
ISBN : 978-93-83328-97-0
Reprint -

Printed By – M/s K.C.Printing & Allied Works, Panchwati, Mathura - 281003

BLOCK INTRODUCTION

In this block, we will learn basic concepts of an operating system and its functions. We will discuss here the evolution of operating systems from early batch systems to modern computer systems. In second unit, we will understand what is a process and relationship between the process and its process control block. This unit includes discussion about a process state and various state transitions a process undergoes. Here, we will also learn distinction between processes and threads along with user level threads and kernel level threads. The third unit will explain us various CPU-scheduling algorithms on which a CPU scheduler is designed. In forth unit, we will understand about the critical-section problem, whose solutions are used to ensure the consistency of concurrent execution of multiple processes. Here, we will be explained mutual exclusion requirements and hardware approaches to support mutual exclusion. In the end of this unit, you will learn a high-level programming language construct called monitor to achieve process synchronization.

UNIT-I INTRODUCTION

Structure

1.1 INTRODUCTION

A Modern computer system consists of processor, main memory, disks, printers, a keyboard, a mouse, a monitor, and various other devices which makes it a complex system as a whole. Managing all these components to use them optimally is a very challenging job. So, computers are equipped with a program called operating system, which acts as an intermediary between a computer user and these computer hardwares to use them efficiently. Most readers have already heard about some of operating systems such as Windows, Linux, FreeBSD, and Mac OS X. A program that provides an interface to users, usually called as the shell (when it is text based) or the Graphical User Interface (when it uses icons). The Shell bypasses user's commands to the operating system to do user’s work. Fig. 1.1 shows an abstract overview of the main components under discussion. Here, the hardware present at the bottom layer on top of which software is present. The hardware consist of: a keyboard, a monitor, disks and other physical devices. Most operating systems have two modes of operation: kernel mode and user mode. The operating system is the most fundamental piece of software which runs in kernel mode (also called supervisor mode). In this mode, it has complete access to all the hardwares, can execute any CPU instructions and can reference any memory address. Kernel mode is generally reserved for the most trusted functions of the operating system. Rest of the software runs in user mode, in which only some parts of machine instructions are available. In particular, those instructions which directly access hardwares, reference memory and control input/ output operation are restricted to user mode programs. The shell or GUI runs in user mode which allows users to start other programs like a Web browser, e-mail reader, or music player. The operating system runs directly on the bare hardware to provide services to all the other software.

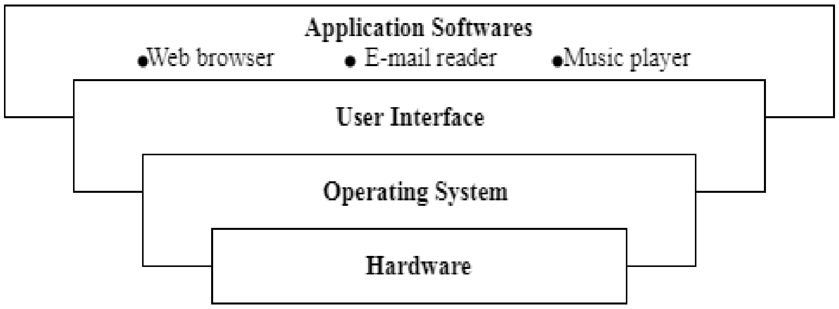


Figure 1.1: Logical view of operating system [3].

1.2 OBJECTIVES

After studying this chapter, you should be able to:

- Explain what operating system is and what it does.
- Describe the evolution of operating systems from early simple batch systems to modern computer systems.
- Summarize the key functions of an Operating System (OS).

1.3 OVERVIEW

A computer system roughly consist of four components: hardware, operating system, application programs and users as shown in Figure 1.2.

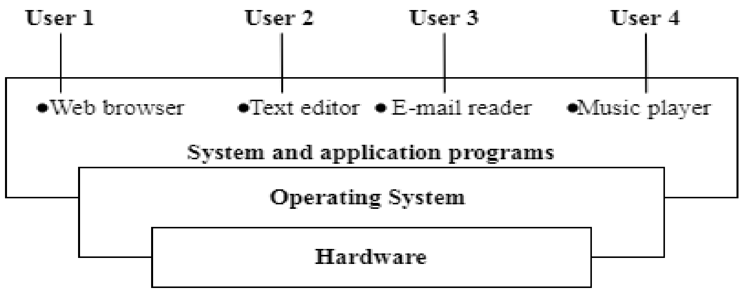


Figure 1.2 - Abstract view of the components of a computer system [1].

The hardware consists of central processing unit (CPU), main memory, input/output devices etc. It provides basic computing resources for the computer system. The Application programs/software like word processors, Web browsers spreadsheets, compilers etc. determine how to utilize these resources to solve user's computation. The operating system controls and co-ordinates the use of hardware among these application programs. Mainly there are two goal of any operating system:

1. **Convenient :** Operating systems should be designed to provide an environment where a user can easily interact (like GUIs) with the computer to execute programs. It should also give fast response to a user request.

2. **Efficient :** Operating system must ensure efficient use of the system resources such as memory, CPU, and I/O devices. System resources will be poorly utilized, if memory or I/0 devices allocated to a program remain idle, while other programs which actually need them are denied access.

1.4 DIFFERENT TYPES OF OPERATING SYSTEMS

Operating systems exists from very first generation of computer and they keep evolving with time. Some important types of operating systems starting from Batch Processing to Multiprocessing system are:

1. **Batch Operating system :** Batch Operating system serves a collection of jobs called as a batch by sequentially reading and executing each job. A job is predefined sequence of commands, programs and data into single unit. The computation in Batch processing is non-interactive because during execution of the batch of jobs, user cannot interact with the system. You present input as a batch of jobs to the system and sometime later you get the output. In batch operating system, the CPU is often idle because the speeds of mechanical I/O devices are slower than CPU.

2. **Multi-programming OS :** One of the important aspect of a modern operating system is to keep the CPU busy all the time. In general a single program cannot makes CPU as well as I/O devices busy all the time. The multiprogramming operating system keeps several jobs in the memory simultaneously as shown in figure 1.3. Every time OS picks one job from the memory and executes it. Meanwhile, a running job may need to wait due to an I/O request. In this situation, any non-multiprogramming OS CPU will sit ideal. But, in multiprogramming OS, the CPU switches and executes another job present in the memory. When the first job finishes its I/O operations, it gets back the CPU. So, the CPU always has one job to execute and it never sits ideal. This way there is good CPU utilization in multiprogramming OS.

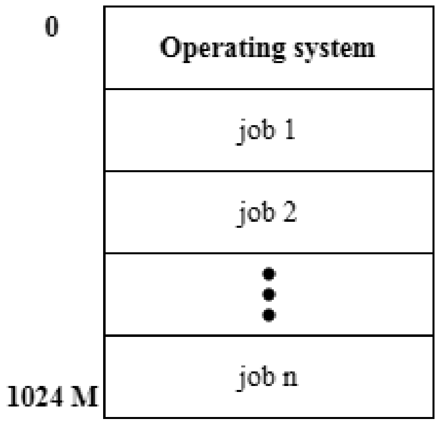


Figure 1.3 : Memory layout for a multiprogramming system [1].

3. Time-sharing OS: Time sharing OS is a logical extension of multiprogramming OS which allows multiple users to use the CPU simultaneously. Each user's action or command in time sharing OS is small that needs only a small CPU time. As the result, the response time is also shorter typically less than one second. The CPU rapidly switches from one user to another (as shown in figure 1.4) giving impression to each user that the entire CPU is dedicated to his/her use, even though it is shared among many users. Both time sharing and multitasking systems require that multiple jobs should be kept in the memory simultaneously. Since, the main memory has limited storage, other jobs are usually kept on hard disk in a job pool. If several jobs are ready to be brought into the memory from the job pool and there is no room to accommodate all of them, a decision is made by the job scheduler to bring among the jobs from the job pool.

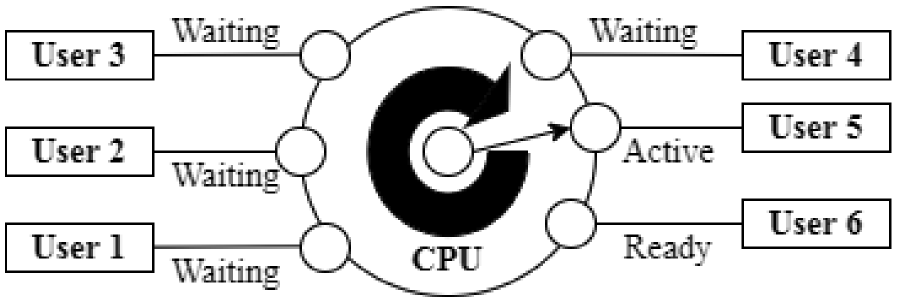


Figure 1.4: In above figure the user 5 is active but user 1, user 2, user 3, and user 4 are in waiting state whereas user 6 is in ready status [3].

4. **Multiprocessing OS :** The main motive of a multiprocessing system is to increase speed of execution in computer system. A multiprocessor system consists of two or more processors operate simultaneously in close communication with one another sharing system bus, sometime clocks, memory and peripheral devices. Multiprocessing systems are of two types: symmetric multiprocessing and asymmetric multiprocessing. In asymmetric multiprocessing, each processor is assigned a specific task by a boss processor. The boss processor schedules and allocates works to other processors. While other processors looks to the boss for instructions. Today, most multiprocessor systems use symmetric multiprocessing (SMP) architecture where two or more identical processors are connected to a single shared main memory each having its own set of registers and local cache, but has full access to all I/O devices. Each processors are controlled by single operating system that treats all processors equally and reserve none for special purposes. The benefit of this model is that it can execute N processes simultaneously with N CPUs. This type of symmetric multiprocessor systems allow processes and resources to be shared among the processors dynamically and can lower the variance among the processors. A recent trend in CPU design

results in a new type of multiprocessing system called a multicore system which contains multiple computing cores on a single chip. Such systems are more efficient than multiple chips each with single processor because on chip communication is faster than between chips communication and also uses less power. As an example, a dual core design with two cores is shown in Figure 1.5.

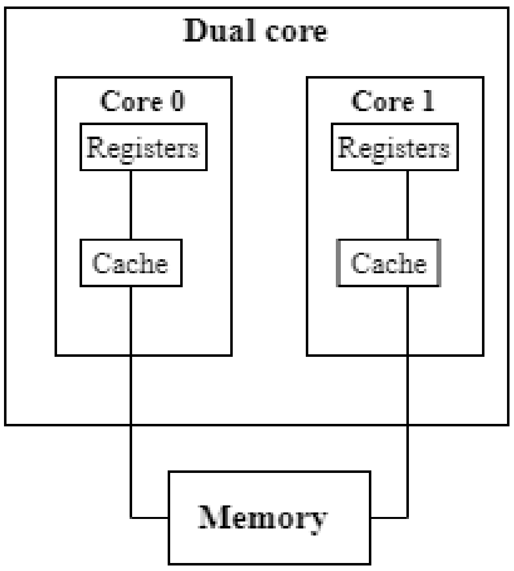


Figure 1.5: A dual-core design with two cores placed on the same chip [1]

These systems increase system reliability by isolating the component which fails and redistribute the computing tasks accordingly.

Check your progress

1. What is the difference between a multiprocessor and a time sharing system?
2. What are the two goals of an OS design?
3. What is the kernel of an OS?

1.5 OPERATING SYSTEM STRUCTURE

Operating system should be designed carefully so that it functions properly and can be easily modified. Generally, OS is partitioned into smaller relatively independent components rather than one individual component. Each of these components has a well-defined function. This makes complexity of the design manageable. In this section, we will discuss typical approaches to design architecture of operating systems. We will see how these components are interconnected and combined into a kernel.

1.5.1 MONOLITHIC SYSTEMS

A Monolithic system implements entries services of OS in kernel mode. Since, both user services and kernel services are implemented in kernel address space, Monolithic system becomes bulky and heavy. Monolithic kernel implements CPU scheduling, memory management, file systems, and all other OS services into a single big unit. Since, all services exist in the kernel address space which can be invoked directly, it is faster than microkernels system. A typical structure of a monolithic kernel is shown in figure 1.6.

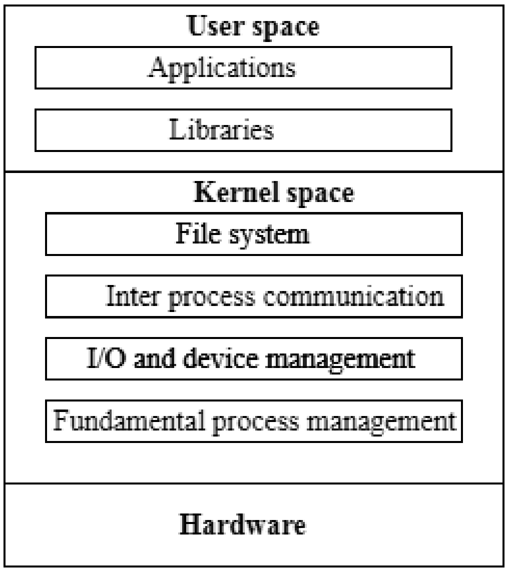

User space
Applications
Libraries
Kernel space
File system
Inter process communication
I/O and device management
Fundamental process management
Hardware

Figure 1.6 Monolithic Kernel [1].

Adding a new feature to a monolithic system requires recompiling of whole kernel, whereas with microkernels you can add new features or patches without recompiling the whole kernel. Earlier, UNIX and Linux are based on Monolithic system and later on at the end of the 1980's the idea of Microkernel was conceived.

Advantages :

1. Since all user and kernel services are invoked through kernel mode, monolithic system is faster than microkernel system.

Disadvantages :

1. The size of these systems are larger than microkernel system.
2. Adding of a new service or feature requires recompiling of the whole system.
3. If any services fails, this lead to failure of the entire system.

1.5.2 MICROKERNEL SYSTEMS

This structure of operating system removes all non-essential services from the kernel, and implement these services as applications services. This system design makes the kernel as small and as efficient as possible. It implements basic process management, memory management, message passing, and other important services in kernel address space. While, other user services are implemented in user address space as shown in figure 1.7. This design enhances security and protection of the system because most services are performed in user mode. The system expansion in Microkernel is easier which requires only adding more services in user address space as application services rather than modifying the kernel. Windows NT was originally microkernel and later on Windows NT 4.0 was introduced with improved performance by moving more services into the kernel.

Advantages :

1. The size of the kernel is small because it contains only core services.

2. System expansion only requires adding new services as application programs instead of rebuilding of a new kernel.

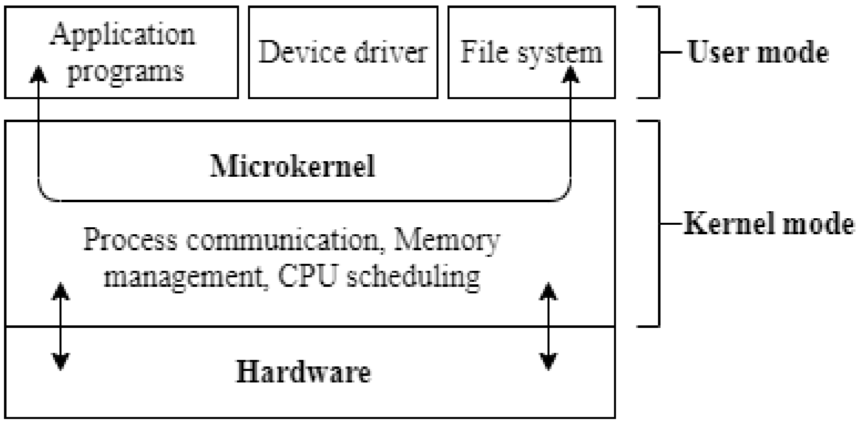


Figure 1.7: Architecture of a typical microkernel [1].

1.5.3 LAYERED SYSTEMS

In Layered approach the OS consists of a number of smaller layers. The bottom layer (Layer 0) is the hardware and the highest layer (N) is the user interface. This layered structure is shown in Figure 1.8. Each layer resides on the layer below it, and depends only on the services provided by just previous lower layer. For example, a particular layer M consists of data structures and a set of routines that is invoked by higher-level layer. The main advantage of the layered approach is simplicity of construction and debugging. Each layer can be developed and debugged independently since all lower layers have already been debugged to deliver proper services. The major challenge with layered design is deciding in what order to arrange these layers because any lower layer cannot call the services of any higher layer.

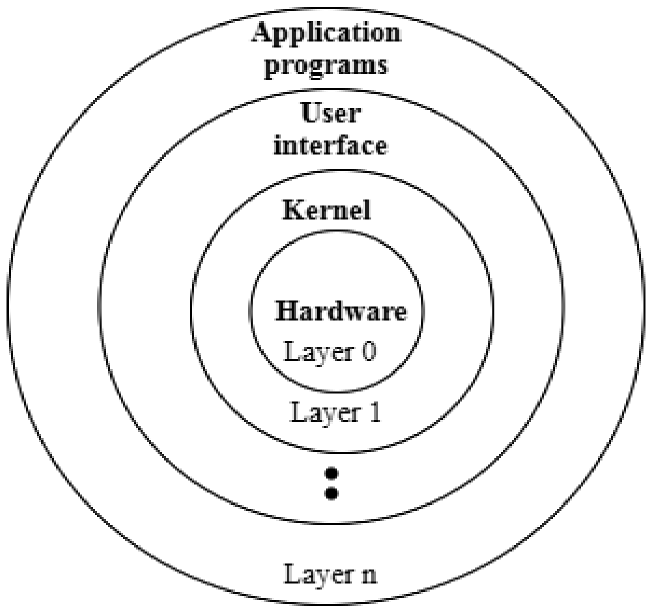


Figure 1.8 : A layered operating system [1].

Advantages :

1. Maintenance of a layer is easy to do without affecting layer interface.
2. It provides modularity and clear interface.

Disadvantage :

1. The performance of layered system is not so good because any request for service from a higher layer has to go through all subsequent lower layers.
2. It is difficult to assign functionalities to appropriate and correct layer.

1.5.4 MODULAR SYSTEMS

The idea of designing this type of system is to provide the kernel a relatively small core services and as the kernel need to run other services, they are dynamically linked through loadable modules. New services can be dynamically added through loadable modules which do not compile or rebuild the kernel every time. This approach to OS design is somewhat similar to layered system where modules are similar to layers which have well defined tasks and interfaces. Here, the modules can directly communicate among each other and thus, this eliminates the problem of going through multiple intermediary layers. It is more flexible than layered approach because the kernel has knowledge of how to load modules. Figure 1.5 shows Solaris operating system which consists of core Solaris kernel organised around 7 loadable modules.

Advantages:

1. The size of the kernel is small because it contains only core services.

2. It has the advantages of microkernel system without message passing overhead because the modules do not require message passing in order to communicate each other.

3. New services can be added through loadable modules which do not require rebuilding the kernel.

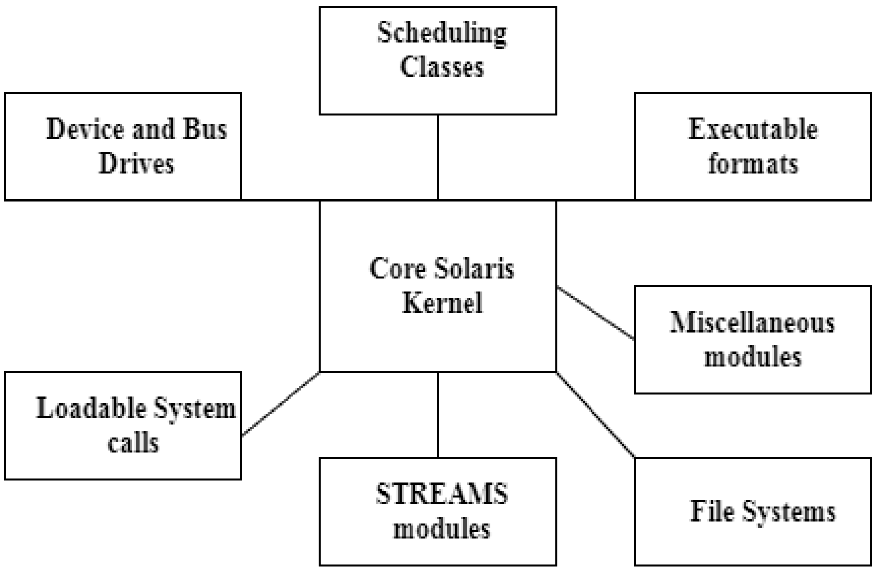


Figure 1.5 : Solaris loadable modules [1].

1.5.5 HYBRID SYSTEM

Most of the modern operating systems actually combines several approaches, rather than one pure model. A Hybrid System combines multiple approaches of OS design to address performance, security and usability needs. Linux and Solaris are monolithic because both user and kernel services are present in single address space (or kernel mode). They are also modular because new services can be added through dynamically loadable modules. Windows are mostly monolithic due to performance reasons and they are microkernel because some of users functionalities are available is user mode. Apple Mac OS X consists of Mac microkernel and BDS kernel. The Mac kernel provides services such as memory management, inter-process communication, message passing and threads scheduling. The BDS kernel provides additionally services including command line interface, networking and file system. The kernel environment also I/O kits for development of device drivers and loadable modules (for kernel extension). Figure 1.6 shows structure of Mac OS X system.

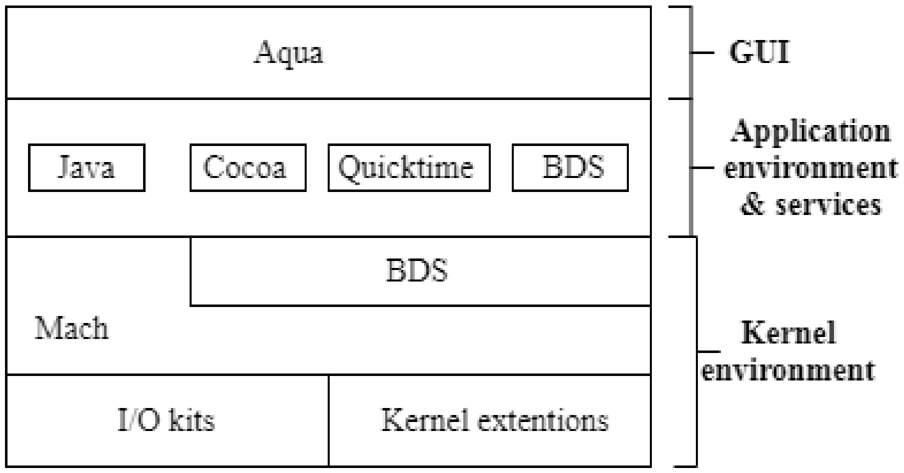


Figure 1.6 : The Mac OS X structure [1].

<u>Check your progress</u>

1. What are the advantages and disadvantages of Monolithic and Microkernel structure of Operating system?
2. Why does the layered system structure suffers from performed issue?
3. How does the modular system provide extension of services?

1.6 FUNCTIONS OF OPERATING SYSTEM

The major functions of operating system are following:

1. **Process management :** A program in execution is called process. For example, a word processing program running by a user on computer is a process. The operating system is responsible for allocating various processes to CPU in a efficient way. OS will block a process which requests for an I/O event and resume its

execution when its I/O event is completed. OS will terminate a process, if its execution is completed. OS provides of inter-process communication that allows a process to share data and information with other processes. The OS also provides a synchronization mechanism to maintain data consistencies caused due to concurrent execution of processes. OS is responsible for management of all these services related to processes.

2. **Memory management :** A process cannot be executed unless it is brought into main memory. In multiprogramming environment, multiple processes reside into the main memory. The memory management responsibility of an OS includes allocating a portion of memory to a new process, deallocating portions of memory when they are freed, keeping track of parts of memory that are in use, swapping processes between main memory and disk when main memory is not enough to hold all the processes.

3. **Storage management :** The OS is responsible for storage of files on different types of storage media such as optical disk and magnetic disk. Each type of storage media has different characteristics and physical organisation which are controlled by their respective devices such as disk drive and tap drive. Each drive has unique characteristics such as transfer speed, access speed, access method and storage capacity. The operating system hides these physical details from users and implements the abstract concept of a file by managing the mass-storage media and the devices that control them. OS organises files into secondary storage by providing various operations such as creating and deleting files and directories.

4. **Disk management :** Operating system is responsible for efficient use of peripheral devices with minimum access time and with large disk bandwidth. This can be achieved by servicing the disk I/O requests in effective order. Generally, the OS serves a request immediately, when the desired disk drive and disk controller is available. But, if the drive or controller is busy, further requests are placed in a disk queue for that drive. In a multiprogramming system, the disk queue often have several pending requests. The OS serves these pending requests in the queue in effective order which would result minimum access time and large disk bandwidth.

5. **Protection and security :** If a computer system has multiple users and allows the concurrent execution of multiple processes, then access to files, memory segments, CPU, and other resources must be controlled. The OS provides protection and security which ensures that OS resources can only be used by the processes after their proper authorization by the OS. Each user in an OS is distinguished by its user ID. Each user is authenticated at the time of system log-in. After their authentication, all associated

processes and threads are accessible that user. OS also implements group ID to distinguish among sets of users rather than individual users. For example, one set of users may be allowed to perform all operations on a particular file, whereas another set of users may only be allowed to only read the file. Each group ID has a set of users belonging to that group.

1.7 SUMMARY

In summary,

- You are introduced to an operating system, which is software that provides interface to user to manage the computer hardware as well as provides an environment for application programs to run.

- You have learnt some of the important types of operating systems starting from Batch Processing to Multiprogramming, Time sharing and Multiprocessing.

- You observed that OS tasks are partitioned into small components rather than have one individual component. Each of these components have well-defined functions, with carefully defined inputs and outputs.

- Finally, you learnt that Process management, Memory Management, Storage Management, Disk Management and Protection and Security are the key functionalities of any Operating System.

1.8 TERMINAL QUESTIONS

- Discuss the essential properties of the following types of operating systems:
  - ➢ Batch
  - ➢ Interactive
  - ➢ Time sharing

- What are the various problems that can arises in a multiprogramming and time-sharing environment, where several users share the system simultaneously?

- Can we ensure the same degree of security in a time-shared machine as in a dedicated machine? Explain your answer.

- Under what circumstances it would be better for a timesharing system rather than a PC that support only one program or a single-user workstation?
- Illustrate the advantages and disadvantage of multiprocessor systems?
- Explain the key differences between symmetric and asymmetric multiprocessing system.
- Discuss the main advantage of the layered approach to system design. How it is different from modular approach?
- Briefly explain the major functionalities offered by an OS.

UNIT-II PROCESS AND THREAD

Structure

2.1 INTRODUCTION

Computers in earlier time only execute one program at a time. The program takes entire control of computer system and its resources. Today, computers are able to execute multiple programs at a time by bringing multiple program into the main memory as process. A process is the basic unit of work in a modern computer system. A modern computer system has multiple process which can be broadly divided into two types: operating system processes and users' processes. The operating system executes system code while user processes executes user code. Operating system executes all these processes concurrently by switching among these processes to make better utilization of CPU.

2.2 OBJECTIVES

In this Unit,

- You will learn a basic concept of a process and its various features.
- You will understand how operating system executes processes
- We will discuss about the concept of a thread which is the fundamental unit is CPU utilization.

2.3 PROCESS CONCEPT

When computer is busy in execution, it performs CPU activities. These activities are sometimes called as jobs in batch system. While, a time sharing system calls these activities as a user program or tasks. In a multitasking system with single user, he/she can access multiple programs at one time such as an e mail, word processor and web browser.

Informally, many times we refer a programs as a process, but a process is more than a program/program code. The process consists of: a program code, the value of program counter, content of processor's registers, process stack, a data section and a heap. The content of the process is also shown in Figure 2.1. The program code of a program is called as text section. The process during execution of the program code contains temporary data such as local variables, function parameters and return addresses. These temporary data are stored in

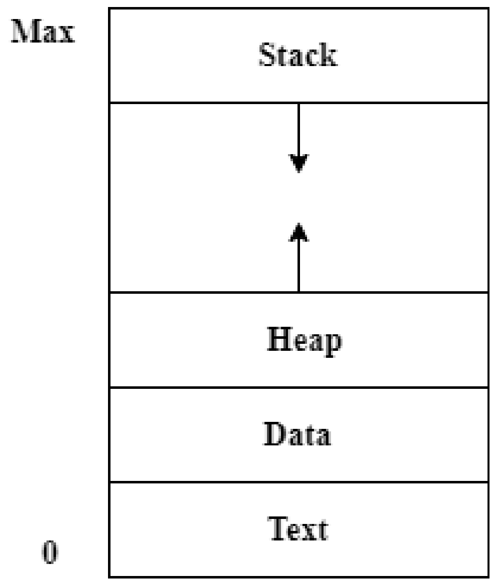


Figure 2.1: Process in memory [1].

The process's stack. The global variables of the program code are stored in data section. The heap is the memory that is dynamically allocated to the process during its execution.

So, we can now say that a program is not a process. It is a passive entity which is just a list of instructions stored on an executable file. While, a process is an active entity which contains a program counter along with several data to represent its current activity. A process is generally executed either by double click or name of the executable file in the command line.

2.4 PROCESS STATES

When a process executes, it undergo a sequence of activities which is divided in parts or states. A process undergoes following states during its execution:

- **New :** A newly created process first enters in New state.

- **Ready :** A process is ready to be allocated to the CPU to start its execution it enters in Ready state.
- **Running :** When the instructions of the program code is executed, it enters in Running state.
- **Waiting :** During execution of the program code, the process requires several events such as I/O operations or computer resources. When these resources is/are not available, the process goes to waiting state.
- **Terminated :** When a process finishes its execution, it reaches to terminate state.

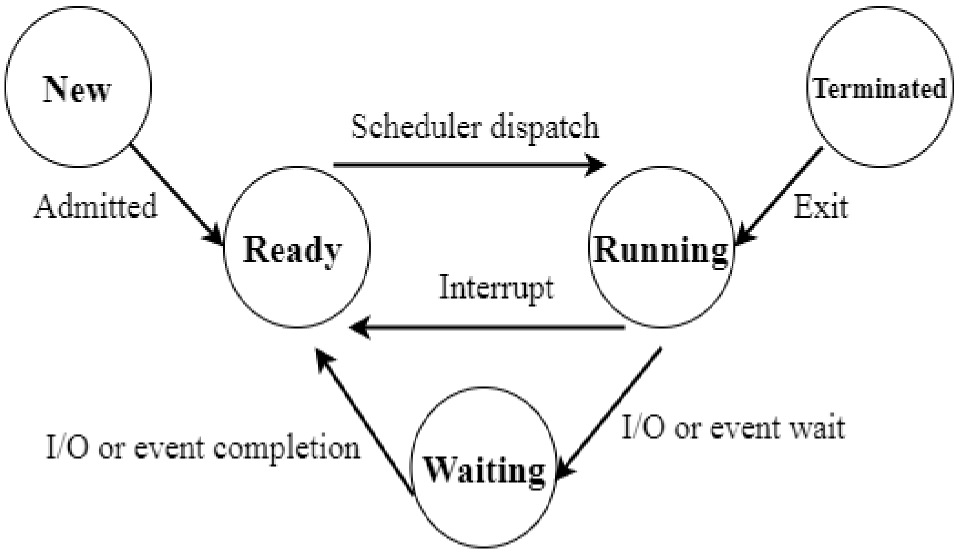


Figure 2.2 : Diagram of process state [1].

The process switches among these states during lifetime of its execution as shown in figure 2.2. A process may undergo through following possible transitions :

- **New – Ready :** When OS wants to increase degree of multiprogramming, it brings several processes from New state to Ready state. The OS set some limit on number of processes in ready queue so that it will not degrade system performance.
- **Ready – Running :** OS must selects a process from the ready queue and allocates it to the CPU to keep the CPU busy all time. When the process gets CPU for its execution it changes its state from Ready to Running state.
- **Running – Terminate :** When the CPU executes a process and finally the process finishes its execution, it changes its state from Running to Terminate state.
- **Running – Ready :** This transition of state happens when a running process has reached the maximum allowable time (time slice) for uninterrupted execution. This transition may also

happens when some high priority process has completed its I/O operations and become available in ready queue for its execution.

- **Running – Blocked :** A currently running process is put in the Blocked state from Running state, if it requests an I/O operation in middle of its execution. At this time, OS must select another process from ready queue and assign it to CPU.
- **Blocked – Ready :** When a process is in the Blocked state completes its I/O operation, it is moved to the Ready state.

Check your progress

- Describe the purpose of a Process Control Block (PCB)?
- What are the differences between Ready queue and the Blocked queue?
- What do you mean by context switch?

2.5 PROCESS CONTROL B LOCK

Operating system recognises each process by their Process Control Block (PCB). A PCB contains many piece of information of a process which is essential for its execution. A typical structure of a process control block is shown in figure 2.3. Generally, a PCB consists of following information:

- **Process Number (Process ID) :** Each process in an operating system is uniquely identified by its process ID. This field contains process ID is the process.
- **Process state :** It indicates the current state of a process. This may be one of the states: new, ready, running, waiting, terminated.
- **Program Counter :** The value of this field is the address of next instruction in the program code to be executed. The value of the program counter changes every time when a new instruction is executed by CPU.
- **CPU registers :** CPU often communicates with registers while executing any instruction. Generally, CPU has multiple registers depending on computer architecture which communicate directly with the CPU. This includes accumulators, index registers, general purpose registers and stack registers. The values of these registers are stored in this field.

- **CPU scheduling information :** The value of this field includes information such as priority of the process and pointer to scheduling queue.
- **Memory- management information :** This field contain information related to memory management such as value of base and limit registers of the page table or segment table.
- **I/O status information :** During execution of a program code, it may require several I/O devices, files etc. This field contains information including list of I/O devices allocated to a process and list of open files during the process execution.
- **Accounting Information :** This includes information like amount of CPU used for a process execution, time limits, process numbers etc.

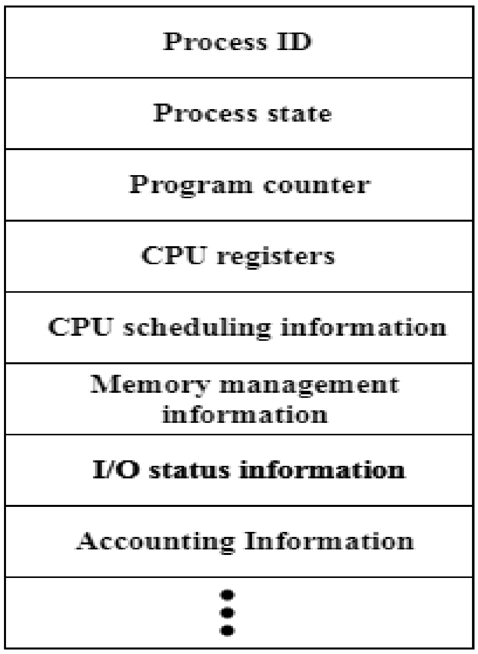


Figure 2.3 : Simplified Process Control Block [4].

2.6 SHORT TERM SCHEDULER, DISPATCHER AND CONTEXT SWITCHING

Short term scheduler : When a process is newly created, it enters to a ready queue. This queue holds all the processes of a system that are ready to execute. But, only one process is selected from the ready queue for execution by CPU. This decision is made by a CPU scheduler or a short term Scheduler which is designed on the based on various scheduling algorithms which we will see in next chapter.

Context Switch : During the execution of the process, it may execute for a while and then it may be interrupted, terminated or waited for some I/O events. As the result, the process will undergo through various states.

When the process is either interrupted or waiting for an I/O event, it must be suspended and some other process from the ready queue must be selected for execution. This requires saving context of the suspended process to resume its execution later and restoring context of the new process to start its execution. The context information of a process are represented by entries in its PCB such as process state, CPU registers and memory management information. So, when a CPU switches from one process to another, it requires saving context of old process in its PCB and restoring context of new process from its PCB. This task is known as context switching. This is purely overhead to a system because during context switching the system does not perform any useful work.

Dispatcher : When a process from ready queue is selected by the short term scheduler, a module called Dispatcher gives a control of a CPU to the process. So, the dispatcher comes into the picture just after the short term scheduler finishes its job. The dispatcher is invoked for every process switch. The time taken by the dispatcher to stop one process from its execution and start another process for execution is called dispatch latency. The dispatcher involves following functions :

1. Switching context
2. Switching to user mode
3. Jumping to the proper location in the user program to restart that program

2.7 THREAD

Threads are very useful in modern operating systems where a process consists of multiple threads to perform multiple independent tasks. If a process consists on multiple threads, then several threads of a process may run at the same time and they can perform different tasks simultaneously. For example, consider a word processor like MS word where one thread checks for grammar and spelling check, second thread takes user input from keyboard and third thread makes periodic backup of current file on disk. Consider another example of a web browser which consists of multiple threads. Here, one thread may be responsible for scrolling of the web page while second thread loads images into the web page. The third thread may play animation and videos.

Process vs. Threads : Processes and threads are similar in many ways. They operate in same way with some differences.

Similarities :

1. Similar to processes, threads also shares CPU and only one thread of a process runs at any time.

2. Like processes, if one thread is blocked, another thread can execute.

3. Like processes, threads can also create child.

Differences :

1. Each process is independent from another, while each thread within a process are not independent from other threads within the same process.

2. Threads within a process are not independent from each other because they share code section, data section and OS resources. While, processes are different from each other since they originate from different users.

2.8 THREAD MODELS

A process is heavyweight which is also known as single threaded process. A thread has some properties of a process, so it is also called as light weight process. A Thread is a basic unit of CPU utilization which contains its own thread ID, program counter, registers and a stack. A thread shares code section, data section and operating system resources like files of other threads in the same process as shown in figure 2.4.

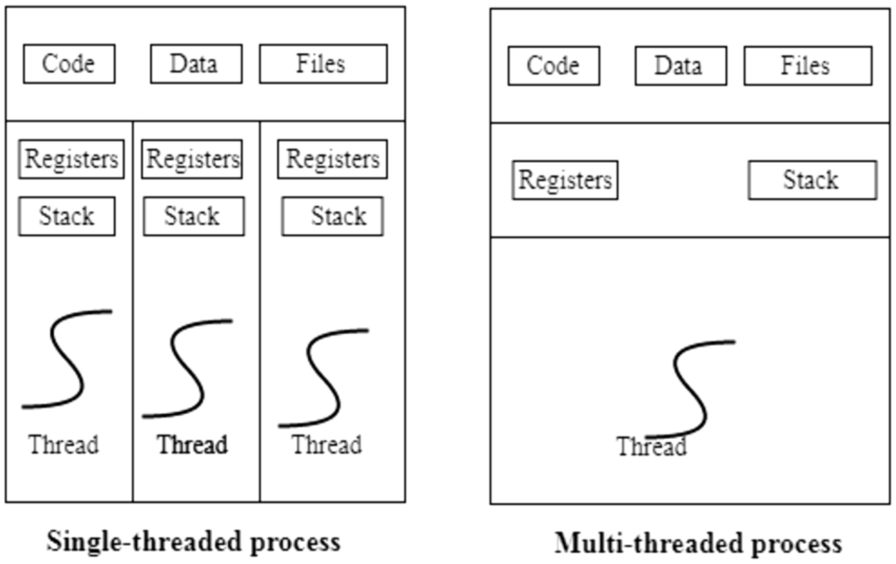


Figure 2.4 - Single-threaded and multithreaded processes [1].

Running multiple threads in a process is similar to running multiple processes in a computer system. The term multithreading is used to describe a situation when a process consists of multiple threads. For example, figure 2.5 (a) shows three traditional processes, each has its own address space and a single thread of control. In figure 2.5 (b), three threads of control are present in a single process. In both cases, three threads are present, but, in figure. 2.5 (a) each thread executes in different address space. Whereas, in Fig. 2.5 (b) all three threads execute in same address space.

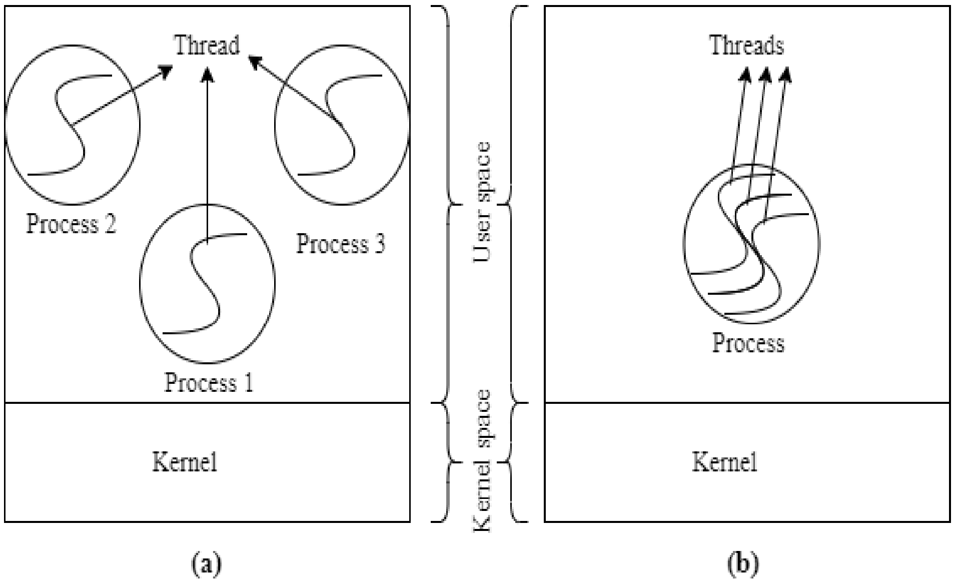


Figure 2.5 : (a) Three processes each with one thread. (b) One process with three threads [3].

The situation shown in figure 2.5 (a) is relevant where these three process are completely unrelated to each other. The organisation shown in figure 2.5 (b) is appropriate where each of these threads are the part of same job and are in close communication with each other. Each thread in a multithreaded process gets CPU one by one and executes. The CPU switches among these threads rapidly and gives illusion that all threads are running simultaneously in parallel. Similar to a process, a thread may go to one of the states: ready, running, blocked, or terminated. A thread in running state is currently active and executed by the CPU. A blocked thread waits for some I/O event to complete. A ready thread is ready to execute but wait for its turn to get the CPU. The transitions between thread states are also same as the transitions between process states.

It is important to note that each thread maintains its own stack, as shown in Figure 2.6. The stack contains one frame for every procedure call called but not yet returned. The frame holds local variables and return address, when a procedure call is finished. For example, consider a thread calls a procedure X which calls another procedure Y. The procedure Y also calls another procedure Z. While the procedure Z is executing, the frames for all three procedure X, Y and Z is stored in the stack. Each thread within a process may call different procedures which results in different execution history. So every thread maintains its own stack.

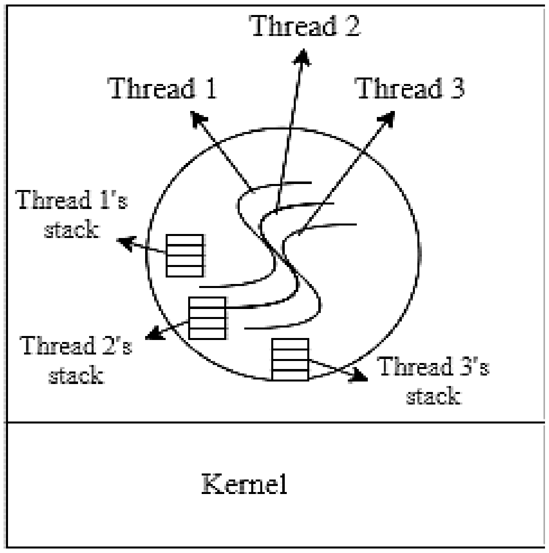


Figure 2.6 : Each thread has its own stack [3].

2.9 THREAD USAGE

The various reasons for using threads over process for different applications are:

- **Responsiveness :** If a process consists of multiple threads, when any thread finishes its execution, its output can be immediately returned to the user. While, a heavyweight process (single threaded process) may take longer time to finish its execution and the user will get late response.

- **Efficient :** Threads take less time to create and terminate. Also, the context switching between two threads are faster than two processes since threads within a process share memory and files without invoking the kernel.

- **Effective utilisation of multiprocessor system :** A multithreaded process consists of multiple threads. Each can be scheduled to run on different processors. This makes faster execution of the process.

- **Increased throughput :** In a multithreaded process, each thread performs a separate job. Due to faster context switching time and more resource sharing, more number of jobs can be completed per unit time. This increases system throughput.

Check your progress

1. How many threads does a traditional, heavyweight process have?
2. Provide at least three benefits of multithreaded programming.

3. What are some real world examples of multithreading in computer?
4. How does a thread is similar and different from a process?
5. Compare and contrast the resources shared by processes and threads.

2.10 IMPLEMENTING THREAD IN USER AND KERNEL SPACE

There are two kinds of threads in modern operating systems: user level threads and kernel level threads. User level threads are implemented by programmers while developing their application or software. This type of threads does not requires any support from operating system. Kernel level threads are implemented within kernel of an operating system. When an operating system supports kernel level threads, it can perform several tasks at the same time like multiple system calls simultaneously.

2.10.1 IMPLEMENTING THREADS IN USER SPACE

In User level thread implementation, entire threads package is placed into the user space and the kernel knows nothing about them. The kernel treats them as an ordinary single threaded processes. A user-level threads can be implemented in an operating system even when it does not support threads. The general structure of threads implementation in User Space
is illustrated in Fig. 2.7(a).

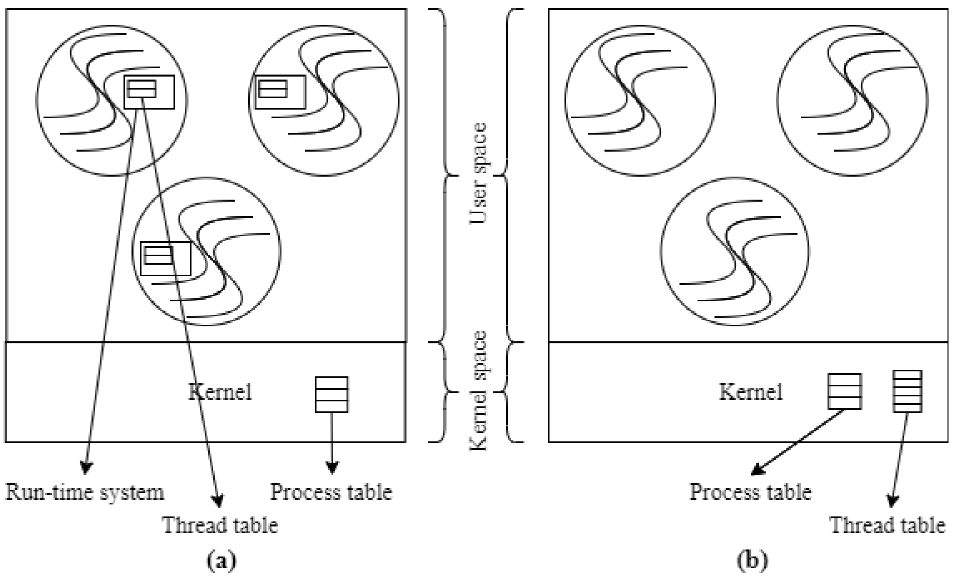


Figure 2.7: (a) A user-level threads package. (b) A threads package managed by the kernel [3].

The threads run on the top of run-time application program, which is a collection of procedures such as thread_create, thread_exit, thread_wait, and thread_yield. All the threads operations such as creating a new thread, switching between threads and synchronization of threads are performed by procedural call without any involvement of the kernel. So, the user level threads are much faster than kernel level threads. When threads are managed in user space, each process maintains a thread table to keep track of its threads. The thread table keeps track of each thread's program counter, stack pointer, registers, state, etc. The thread table is managed by the run-time system. When a thread is moved to blocked state, the current state information is saved in the thread table so that it can be restart again in future. When a waiting thread finishes its I/O event it calls a run-time system procedure to reload the thread's registers, PC, SP from the thread table. As soon as machine registers, stack pointer and program counter have been reloaded with new values, the waiting thread enters into running state.

Advantages :

1. User level threads can be implemented in any operating systems even if they do not support threads.
2. Fast and efficient: Threads switching in User level threads do not take much time than the procedural call. So they are faster than kernel level threads.

Disadvantages :

1. Since, the user level threads are invisible to kernel, operating system can make poor scheduling decision such as blocking a process even it had other runnable threads, scheduling a process having idle threads.
2. Operating system allocates one time slice to each process whether it contains one threads or 100 threads.
3. If one thread within a process is blocked, the entire process is blocked even the process has all other threads runnable.

2.10.2 IMPLEMENTING THREADS IN KERNEL SPACE

Kernel Level Threads Implementation: In this method, operating system knows about the threads. So, all the thread operations are implemented in kernel and operating system schedules and manages threads through system calls. There is no thread table for each process and run-time system are needed as shown in Fig. 2.7(b). Instead, the kernel maintains its own thread table that keeps track of all the threads in the system. The thread table holds thread's registers, state, and other information for each thread in the system. A thread can create a new thread or destroy an existing thread through a kernel call. The creation or destruction of a thread requires updating the kernel thread table. All calls which block a thread are implemented as system calls, at considerably greater cost than a call to a run-time system procedure in case of user level threads. When a thread is blocked, the kernel can schedule to run another thread either from same process or from different process. While, in case of user-level threads, the run-time application keeps running threads from its own process until the time slice expires or there are no runnable threads left to run.

Advantages :

1. Since the kernel has full knowledge of all threads in the system, it may allocate more CPU time to a process having more number of threads than other processes.
2. If one thread in a process is blocked, then other threads in the process may continue to run and the whole process will not be blocked.

Disadvantages :

1. Because the kernel is responsible for managing both threads and processes, there is significant overhead to the kernel.
2. Kernel threads are slower than user level threads.

2.11 SUMMARY

In summary

- We learnt that a process is a program in execution. As a process executes, it changes state.

  Each process may be in one of the following states: new, ready, running, waiting, or terminated. Each process is represented in the operating system by its process-control block (PCB).

- We saw that there are two major classes of queues in an operating system: Blocked queues and the ready queue. The ready queue contains all the processes that are ready to execute and are waiting for the CPU. Blocked Queue holds all the processes in blocked state.

- We learnt that a context switch requires the kernel to saves the context or state of the running process into its PCB and loads the saved context of the new process scheduled to run.

- We saw that a multithreaded process contains multiple threads within the same address space, each having their own program counter, stack and set of registers, but sharing common code, data, and certain structures such as open files. If a process has multiple threads of control, it can perform more than one task at a time.

- Finally we learned that there are two main ways to implement a threads package: in user space and in the kernel. User-level threads are managed by the run-time system, which is a collection of procedures. While kernel-level threads are managed by kernel system calls.

2.12 TERMINAL QUESTIONS

1. Describe the actions taken by a kernel to context-switch between processes.
2. Describe the purpose of a Process Control Block (PCB)?
3. What is the role of the process scheduler?
4. What are the possible transitions that a process may undergo?
5. Illustrate the differences between Ready queue and the Blocked queue.
6. Discuss the two approaches for implementing a thread library.
7. What are the advantages and disadvantages of user-level threads over kernel-level threads?
8. How the User level and kernel level threads are implemented?

UNIT-III PROCESS SCHEDULING

Structure

3.1 INTRODUCTION

In a multiprogramming environment, there are multiple processes in a ready queue competing for CPU for their execution. So, a decision must be made to select a process from the ready queue. The part of the operating system which makes such decision is called short term scheduler or CPU scheduler. When a new process enters the system, a scheduling decision must be made whether to continue the execution on currently running process or starts the execution of newly created process if it has high priority. When a process finishes its execution and exits from the system, a scheduling decision again must be made to select some other process from ready queue for its execution. When a process is blocked on an I/O event, a scheduling decision is made to pick some other process for its execution. Also, when a process completes its I/O event, the CPU scheduler must make a scheduling decision whether to start the execution of the process or schedule some other process. The CPU scheduler is designed on the basis of various scheduling criteria such as CPU utilisation, system throughput, response time and waiting time. There are various types of scheduling algorithms exist based on these scheduling criteria which we will discuss in this chapter.

3.2 OBJECTIVES

After studying this chapter, you should be able to:

- Describe the need and criteria of CPU scheduler.
- Understand various CPU-scheduling algorithms.

3.3 CPU-I/O BURST CYCLE

Nearly all processes execute by alternating between CPU-burst and I/O request as shown in Figure 3.1. Typically, a process runs for a while without stopping and then request I/O event such as read or write from a file. When the I/O event is completed, the process executes again until it need more I/O operations. Some processes run similar to as shown in figure 3.1 (a) are CPU-bound processes which spend most of their time in computing. While other type of processes run similar to Figure 3.1-(b) are I/O-bound process which spend most of their time waiting for the I/O events. The CPU-bound processes have long CPU bursts and infrequent I/O waits. While on other hand, I/O- bound processes have short CPU bursts and frequent I/O waits.

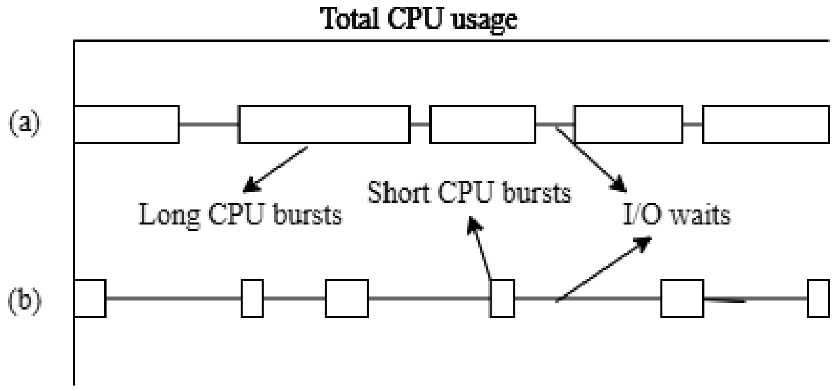


Figure 3.1: Bursts of CPU usage alternate with periods of waiting for I/O (a) CPU-bound process (b) I/O-bound Process [1].

An extensive study shows that, generally any process during its execution has a large number of short CPU bursts and a small number of long CPU bursts (as shown in figure 3.2). An I/O-bound process typically has many short CPU bursts, while a CPU-bound process might have a few long CPU bursts. This distribution can be important in the selection of an appropriate CPU-scheduling algorithm.

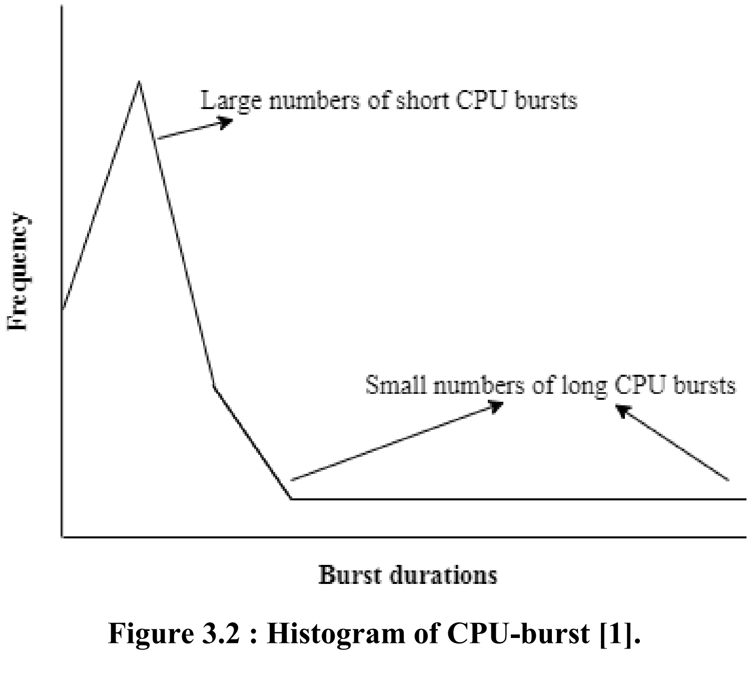


Figure 3.2 : Histogram of CPU-burst [1].

3.4 SCHEDULING CRITERIA

There are several criteria that must be considered, when we compare different CPU scheduling algorithms and select best algorithm for a particular situation.

1. **CPU utilization –** It is the percentage of time that the CPU is busy in execution. Ideally, we want CPU to be busy 100% of the time so that there is 0 wastage of CPU cycles.
2. **Throughput –** It is the number of processes completed per unit time. The throughput may range from 10 processes per second to 1 process per hour depending on the process and specific situation.
3. **Turnaround time –** It is the time from submission of a process to its completion. In other words, it is the total amount of time taken by a process to finish execution which includes amount of time waiting in the ready queue, executing on the CPU, and doing I/O.

   Turnaround Time=Completion Time – Arrival Time

   Or, TAT = CT - AT
4. **Waiting time –** It is the amount of time spent by a process in the ready queue for waiting its turn to get the CPU. It is a difference of turnaround time and burst time (actual CPU time required to execute) of a process.

   Waiting Time = Turnaround Time- Burst Time

   Or, WT=TAT - BT
5. **Response time –** A process arrived in the ready queue will give its first response to us when it will be scheduled to run. In other words, when a process arrived in the ready queue, after how much time it is allocated CPU to run for the first time is a response time.

   Response Time = First Response – Arrival Time

   Or, RT= FR - AT

In general, we want to maximize CPU utilization and throughput while, we want to minimize turnaround time, waiting time, and response time. From a user's point of view, response time is generally most important, while from a system point of view, throughput or processor utilization is important.

3.5 PRE-EMPTIVE AND NON-PRE-EMPTIVE SCHEDULING

There are two general category of CPU scheduling algorithms:

Nonpreemptive : Under nonpreemptive scheduling, once a process is allocated to the CPU, it continue to run until it does not complete its burst time. This scheduling method was used earlier by Microsoft Windows 3.x. Later, in Windows 95 and all subsequent versions of Windows operating systems, preemptive scheduling is used.

Preemptive : Under preemptive scheduling, a running process may be also forced to release the CPU even though it is neither completed nor blocked. In other words a process in running state may be interrupted and moved to the Ready state. The decision to preempt may be performed when clock interrupt occurs i.e. time quantum or time slice expires; a new process arrives or when an interrupt occurs that places a blocked process to the Ready state.

Preemptive Scheduling policies causes greater overhead to OS than nonpreemptive ones. But, the preemptive scheduling provide fair opportunity of execution to all system processes by preventing any one process to hold CPU for long time.

<u>Check your progress</u>

1. Why is it important for the scheduler to distinguish I/O-bound programs from CPU-bound programs?
2. A CPU-scheduling algorithm determines an order for the execution of the processes in ready queue. Given n processes to be scheduled on one processor, how many different schedules are possible?
3. What is the difference between turnaround time and response time?
4. What is the difference between preemptive and nonpreemptive scheduling?

3.6 SCHEDULING ALGORITHMS

3.6.1 FIRST-COME, FIRST-SERVED SCHEDULING

In this Scheduling scheme, a process which arrives first in the ready queue is allocated first to the CPU. This scheduling policy allocates CPU to different processes on the "first come first serve" basis. It is a non preemptive scheduling algorithm. The FCFS policy can be easily implemented by FIFO queue. When a process enters into the ready queue, its PCB is linked at the end of the queue and a process at the head of the ready queue is allocated first to the CPU.

Disadvantages :

1. Starvation is possible in FCFS scheduling because processes at the tail of the ready queue are unlikely to get CPU for its execution.

2. It may cause low CPU and I/O device utilization because a set of processes which requires a I/O device for short time may wait until one process that holds the I/O device for long time finishes its execution. During this time all other process will be blocked.

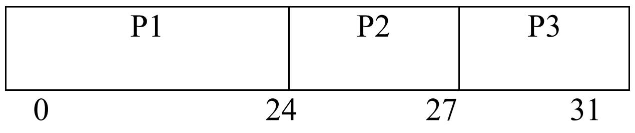
Explanatory Question : Consider the following table of arrival time and burst time for three processes P1, P2 and P3.

Process	Arrival time	Burst Time
P1	0 ms	24 ms
P2	0 ms	3 ms
P3	0 ms	4 ms

The pre-emptive shortest job first scheduling algorithm is used. Scheduling is carried out only at arrival or completion of processes. What is the average waiting time for the three processes if the processes are arrived in the order P1, P2, P3?

Answer: If the processes are arrived in the order P1, P2, P3 in the ready queue, and are served in FCFS order, we get following Gantt chart, which is a bar chart that illustrates a particular schedule, including the start and finish times of each participating process:

Gantt Chart :



Process P0 is allocated processor at 0ms and run for 24ms. At 24ms P0 finish its execution and P2 is allocated to CPU. P2 runs for 3ms and finish its execution at 27ms. At this time P3 allocated to CPU and run for 4ms.

Turn Around Time(TAT) = Completion Time(CT) – Arrival Time(AT)

TAT for P1 = 24 – 0 = 24

TAT for P2 = 27 – 0 = 27

TAT for P3 = 31- 0 = 31

Waiting time = Turn Around Time(TAT)- Burst time

Waiting time for P1 = 24 – 24 = 0

Waiting time P2= 27 – 3 = 24

Waiting time P3= 31- 4 = 27

Average waiting time = (0+24+27)/3 = 17ms

3.6.2 SHORTEST-JOB-FIRST SCHEDULING

This Scheduling algorithm selects the process for execution from the ready queue which has the shortest CPU burst time (CPU time). This algorithm is also called as “Next Shortest CPU Burst Time First” because it allocates CPU to processes by examining the length of next CPU burst (or remaining burst time) of processes rather than its total length burst time. If two processes have same CPU burst time then the process which arrives first in the ready queue is served to CPU. SJF algorithm is of two types: preemptive or non-preemptive SJF. When a new process arrives into the ready queue with a short CPU burst time than the currently executing process, then the preemptive SJF algorithm will preempt the currently executing process and allocates the CPU to the newly arrived process. While on other hand, a non-preemptive SJF algorithm will allow the current running process to finish till its total CPU burst. The preemptive SJF algorithm follows a greedy approach to schedule processes in optimal way. Preemptive SJF scheduling is also known as Shortest Remaining Time first (SRTF) scheduling.

Disadvantage :

1. The algorithm may cause starvation, if shorter processes keep coming. However, this problem can be solved using the concept of aging where the priorities of waiting processes are gradually increased.

Explanatory Question : Consider the following processes, with the arrival time and the CPU burst time given in milliseconds.

Process	Arrival Time	Burst Time
P1	0	6
P2	0	8
P3	0	7
P4	0	3

What is the average turnaround time for these processes with the non pre-emptive shortest remaining processing time first algorithm?

Answer : If the processes are scheduled with non pre-emptive shortest remaining processing time first algorithm, we get following Gantt chart showing the start and finish times of each participating process:

Gantt Chart:

P4	P1	P3	P2

0 3 9 16 24

At 0ms all Processes P1,P2,P3,P4 are arrived in the ready queue but Process P4 is allocated CPU since it has less CPU burst than P1,P2,P3. Process P4 runs for 3ms and finishes at 3ms. At 3ms Process P1 is allocated CPU as it has next shortest CPU burst than remaining processes P3, P2 in ready queue. P1 runs for 6ms and finishes at 9ms at which P3 is scheduled to run. P3 runs for 7ms and finishes at 16ms. Finally at 16ms P2 is scheduled to CPU and runs for 8ms and finishes at 24ms.

Turn Around Time(TAT) = Completion Time(CT) – Arrival Time(AT)

TAT for P1 = 9 – 0 = 9

TAT for P2 = 24 – 0 = 24

TAT for P3 = 16- 0 = 16

TAT for P4 = 3 – 0 = 3

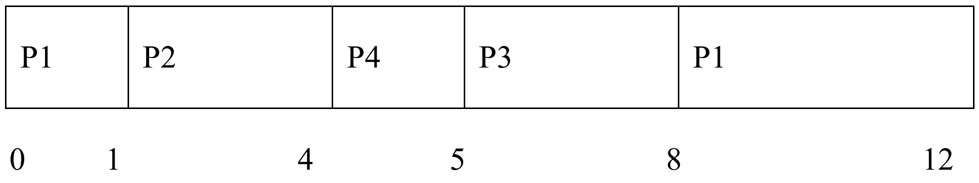
Hence, Average TAT = Total TAT of all the processes / no of processes = (9 + 24 + 16 + 3) / 4 = 52/ 4 = 13ms

Explanatory Question : Consider the following processes, with the arrival time and the length of the CPU burst given in milliseconds. The scheduling algorithm used is pre-emptive shortest remaining-time first.

Process	Arrival Time	Burst Time
P1	0	5
P2	1	3
P3	2	3
P4	4	1

What is the average turnaround time and individual response time of these processes in milliseconds?

Answer : Pre-emptive Shortest Remaining time first scheduling, i.e. that processes will be scheduled on the CPU which will be having least remaining burst time (required time at the CPU). The processes are scheduled and executed as given in the below Gantt chart.



In above Gantt chart Process P1 is allocated CPU at 0ms as there is no other process in ready queue. P1 is pre-empted after 1ms as P2 arrives at 1ms and burst time for P2 is less than remaining burst time of P1. P2 runs for 3ms. P3 arrived at 2ms but P2 continued as burst time of P3 is longer than remaining burst time of P2. After P2 completes at 4ms, other remaining processes P3, P4 are arrived in ready queue. So at 4ms P4 is scheduled as the remaining burst time of P4 is less than the remaining burst time of P1 and p3. At 5ms P4 finishes and P3 is scheduled to run as the burst time of P3 is less than remaining burst time of P1. The Process P3 run for 3ms and finishes as it continue to have shortest remaining burst time than P1. At 8ms, only P1 remains in ready queue, so it is scheduled to run for 4ms after which it finishes its execution.

Turn Around Time(TAT) = Completion Time(CT) – Arrival Time(AT)

TAT for P1 = 12 – 0 = 12

TAT for P2 = 4 – 1 = 3

TAT for P3 = 8- 2 = 6

TAT for P4 = 5 – 4 = 1

Hence, Average TAT = Total TAT of all the processes / no of processes = (12 + 3 + 6 + 1) / 4 = 22/ 4 = 5.5ms

Response Time = First scheduled time – arrival time

Response time for P1=8-0=8ms

Response time for P2=1-1=0ms

Response time for P3=5-2=3ms

Response time for P4=4-4=0ms

3.6.3 PRIORITY SCHEDULING

Priority scheduling is a general case of SJF scheduling where each job is assigned a priority based on burst time and the job with shortest burst time gets a highest priority. Priority scheduling can be either preemptive or non-preemptive. Priorities are implemented using positive integers within a fixed range. The low numbers are used for high priorities and 0 for the highest priority. A Process with the highest priority is executed first and so on. Processes with same priorities are executed on first come first served basis. Priorities of the processes can be decided on the basis of importance of the processes such as average burst time, ratio of CPU to I/O activity and system resource.

Disadvantage : Priority scheduling can suffer from indefinite blocking or starvation of processes, where a low-priority process waits forever because there are always some high priority processes. A common solution to this problem is aging, in which priorities of the low priority processes increase as they wait longer in the ready queue and eventually get their priorities high enough that they can run.

Explanatory Question : Let us consider a set of processes P1, P2, P3 having priorities ranging from 1 to 3. Let us assume that 1 is the highest priority whereas 3 is the least priority. What is the average waiting time in non-preemptive scheduling scheme?

process	CPU Burst Time	Priority	Arrival
P1	10	3	0
P2	5	2	1
P3	2	1	2

The Gantt chart is shown below:

P1	P2	P3	P2	P1

0 1 2 4 8 17

Waiting time = Completion Time(CT) – Arrival Time(AT) – Burst time

Waiting time for P1= 17- 0-10 =7

Waiting time for P2= 8- 1-5 =2

Waiting time for P3 = 4-2-2=0

Average Waiting Time = (7+2+0) / 3 = 3 Millisecond

Here, the preemption is based on the priority when P1 executes, P2 arrive at 1ms with priority 2, which is higher than priority of P1, and thus P1 is pre-empted and P2 get executed. Similarly when P2 execute, P3 arrives at 2ms with Priority 1 which is higher than priority of P2 and thus P2 is preempted and P3 get executed for 2ms and finishes its execution. At 4 ms remaining process P2 have has higher priority than P1 and thus P2 executed for its remaining burst time. And lastly at 8ms P1 get executed for its remaining burst time.

3.6.4 ROUND-ROBIN SCHEDULING

This algorithm can be thought as preempted FCFS scheduling where each processes is given access to the CPU for a fixed CPU time (time quantum) on first come first serve basis. For example, the time quantum may be 20 milliseconds. After this time quantum expired, a running process is preempted and added to tail of the ready queue. The ready queue is managed as a FIFO queue and treated as a circular. A currently running process may also block itself before its time slice expires. If, there are n processes on the ready queue and the time quantum is q, then each process gets 1/n time on the CPU with at most q time units as a whole. And no process waits for more than (n-1)q time units. The choice of the time quantum (q) is extremely important because:

1. If q is very large, Round Robin converts into FCFS.
2. If q is very small, it leads to more the context switch overhead.

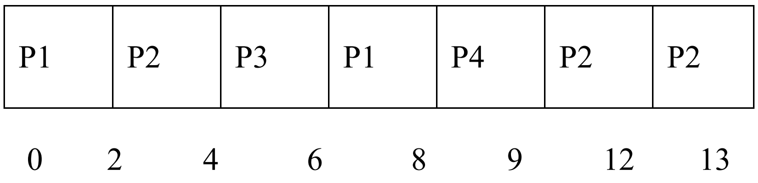
Disadvantage : The disadvantage of this scheduling scheme is the overhead of context switching since there are more context switches in this scheduling.

Explanatory Question : Calculate the average Turnaround time and average waiting time of the processes on the basis of round robin scheduling algorithm. Assume Time Quantum is set to 2 units.

Process	Arrival Time	Burst Time
P1	0	4
P2	1	5
P3	2	2
P4	3	1

Answer : The result of R-R schedule is as depicted in the following Gantt chart:

Ready State : P1 P2 P3 P1 P4 P2P2



According to R-R scheduling we keep the ready queue as a FIFO queue where processes are executed in FCFS. So, firstly P1 arrived in ready queue and is executed first for the time quantum or time slice of 2ms, after which it is pre-empted as new processes P2, P3 is arrived in the ready queue. P1 is added at the tail of ready queue i.e. after P2, P3. Now P2 which is at the head of ready queue is scheduled for execution for the time quantum. At 4ms in Gantt chart P4 is already arrived and added at the tail of ready queue. At this time P2 is pre-empted and added at the tail of ready queue i.e. after P4. Now P3 which is at the head of ready queue allocated CPU and executed for time quantum of 2ms and finishes its execution. At the time of 6ms in Gantt chart, P1 which is at the head of ready queue is scheduled to run for time quantum, after which it finishes its execution. At this time P4 is at the head of ready queue, so it is scheduled to run but its execution completes before the time quantum. So, P4 will execute only for remaining 1 unit of burst time. At this time only P2 is left in ready queue. So again P2 starts its execution for remaining part of burst time and gets executed for time quantum of 2ms and then it is again pre-empts and scheduled again for remaining burst time of 1ms.

Waiting time = Turn Around Time(TAT)- Burst time

Or

Waiting time = Completion Time(CT) – Arrival Time(AT) – Burst time

Waiting time for P1 = 8 – 0- 4 = 4

Waiting time P2= 13 – 1-5 = 7

Waiting time P3= 6- 2-2 = 2

Waiting time P4= 9-3-1 = 5

Average waiting time = (4+7+2+5)/4 = 4.5ms

Check your progress

1. How the round-robin scheduling does differs from FCFS scheduling.
2. Consider three CPU-intensive processes, which require 10, 20 and 30 time units and arrive at times 0, 2 and 6, respectively. How many context switches are needed if the operating system implements a shortest remaining time first scheduling algorithm? Do not count the context switches at time zero and at the end.

3.6.5 MULTILEVEL QUEUE SCHEDULING

A Multilevel Queue Scheduling algorithm partitions the ready queue into multiple separate queues. Each queue holds different types of processes, for instance, one queue may be for foreground (or interactive) processes and another for background (batch) processes as shown in Figure 3.3

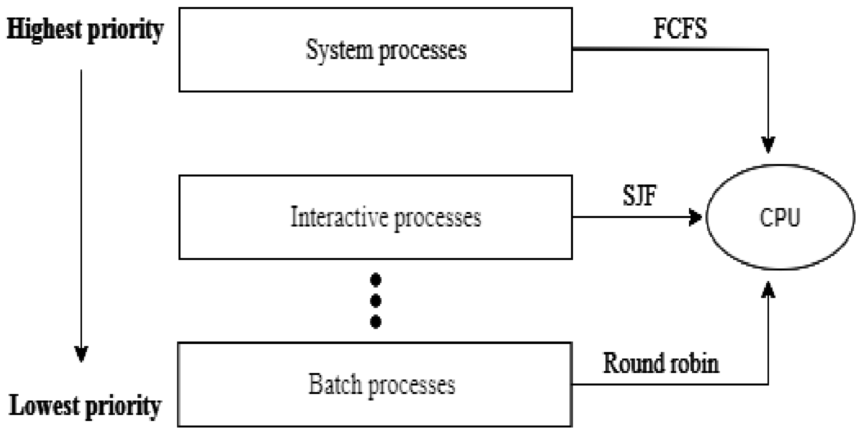


Figure 3.3 : Multilevel Queue [1].

Each queue has its own scheduling algorithm which may be either Round Robin, FCFS, SJF etc. In addition, queues are scheduled via a priority scheduling which serves the first queue with highest priority and so on. Processes in a lower priority queue do not run until all higher priority queues are get emptied. Each queue gets a varying amount of CPU time to schedule its processes. For example, 50% of CPU time may be assigned to the highest priority queue, 20% of CPU time to the second queue, and so on. They may be different time quantum for different queues. When a new process arrives in the system, there is need to specify which queue the process will be put. Note that under this scheduling scheme, processes are permanently assigned to a certain queue and they cannot switch from one queue to another during their execution.

Disadvantage :

It suffers from starvation problem, where a low-priority process waits forever because there are always some high priority processes to execute.

Explanatory Question : Consider a system with Multilevel queue scheduling having four processes: P1, P2, P3, P4. The arrival, burst time and the queue of each process is shown in below table of four processes under Multilevel queue scheduling.

Process	Arrival Time	Burst time	Queue Number
P1	0	4	1
P2	0	3	1
P3	0	8	2
P4	10	5	1

The queue numbers also denote Priority of queues. For example priority of queue 1 is greater than queue 2. The queue 1 uses Round Robin with Time Quantum = 2 and queue 2 uses FCFS scheduling. Calculate the average Turnaround time and average waiting time of the processes under above scheduling scheme.

Solution :

Gantt chart :

P1	P2	P1	P2	P3	P4	P3

0 2 4 6 7 10 15 20

Initial at time 0, queue 1 has processes P1, P2 and queue 2 has process P3. Since, queue 1 has higher priority than queue 2, the processes P1, P2 in queue 1 will execute first. These processes will execute for 7 units of time in a round robin fashion with time quantum of 2 units. After 7 unit of time they finish their execution and queue 1 will become empty. Then, process P3 in queue 2 will execute for only 3 units of time, after which a newly arrived process P4 in queue 1 will start its execution. The process P4 will execute for its entire burst of 5 units and then queue 1 will get emptied. After that, process P3 will execute for its remaining burst time of 5 units.

3.6.5 MULTILEVEL FEEDBACK QUEUE SCHEDULING

In a multi-level queue-scheduling algorithm, we have seen that processes are permanently assigned to various queues. Multilevel Feedback Queue Scheduling allows processes to move among different queues. In this scheduling scheme, If a process in a high priority queue consumes too much CPU time, it is moved to a lower priority queue. And, if a process waits too long in a lower-priority queue, it is moved to a higher-priority queue. For example, consider a Multilevel Feedback Queue with three queues: Q0 –Round Robin Scheduling with time quantum of 8 milliseconds, Q1 – Round Robin Scheduling with time quantum of 16 milliseconds and Q2 with First Come First Serve Scheduling is shown in Figure 3.4.

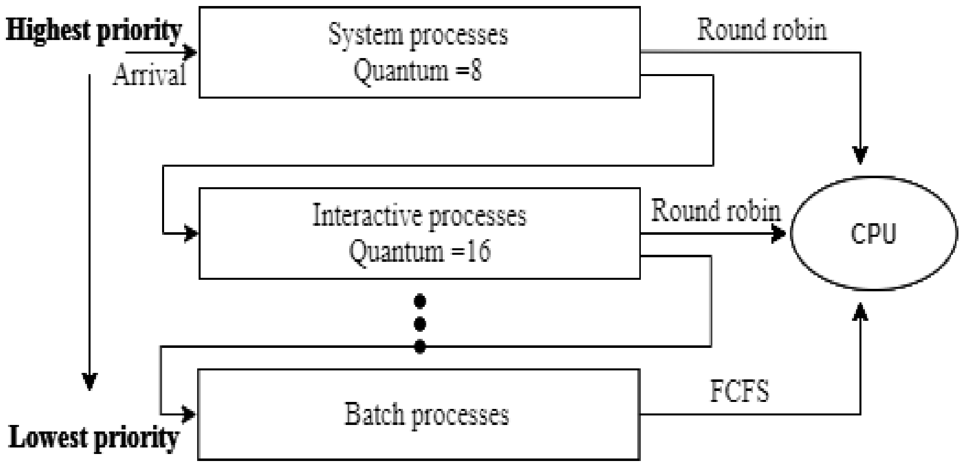


Figure - 3.4 Multilevel Feedback Queue Scheduling [1].

A queue Qi has higher priority than Qi+1. A newly created process enters the queue Q0 where it gets 8 milliseconds to finish its execution. If, the process does not finish its execution in 8 milliseconds, it is moved to the queue Q1 where it gets 16 additional milliseconds to complete its execution. If, it still does not complete its execution, it is moved to the queue Q2 where it is served in FCFS order and finish its execution.

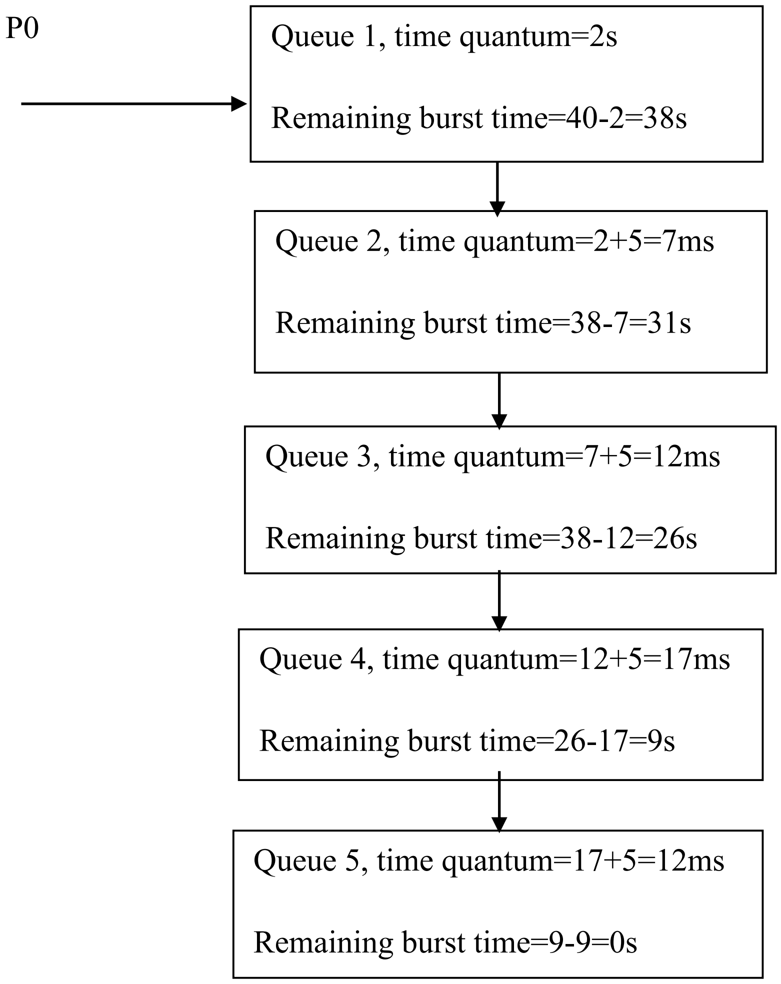
The working of a Multilevel feedback queue scheduler depends on the following parameters:

1. Number of queues.
2. Scheduling algorithms for each queue.
3. Method used to determine when to upgrade priority of a process.
4. Method used to determine when to demote priority of a process.
5. Method used to determine which queue a process will enter initially.

Explanatory Question : Consider a system using multilevel Feed Back Queue scheduling algorithm has a CPU bound process P0 which require the burst time of 40 seconds. The scheduling algorithm is using a time quantum of '2' seconds which is incremented by '5' seconds at each queue. How many times the process will be interrupted during its execution and on which queue the process will terminate?

Solution :

Initially the process P0 enters the queue 1 with burst time 40.


P0
Queue 1, time quantum=2s
Remaining burst time=40-2=38s
Queue 2, time quantum=2+5=7ms
Remaining burst time=38-7=31s
Queue 3, time quantum=7+5=12ms
Remaining burst time=38-12=26s
Queue 4, time quantum=12+5=17ms
Remaining burst time=26-17=9s
Queue 5, time quantum=17+5=12ms
Remaining burst time=9-9=0s

3.7 SUMMARY

In summary,

- We learns various criteria for designing the short-term scheduler. From a user's point of view, response time is generally the most important characteristic of a system, while from a system point of view, throughput or processor utilization is important.

- Finally we learnt variety of CPU Scheduling algorithms for making scheduling decision among processes in ready queue.

  First-come-first-served : Select the process for execution that arrives first in the ready queue.

  Round robin : Use a time quantum to limit any running process for a short CPU time, and rotate short CPU time execution among all processes in ready queue.

  Shortest remaining time : Select the process with the shortest remaining CPU time. A process may be preempted when another process with the shortest CPU time arrives in ready queue.

  Priority scheduling : Each process is assigned a priority where a process with the highest priority is executed first and so on.

  Multilevel Queue Scheduling : Partitioned the Ready queue into multiple separate queues and allocate processes to these queues based on execution history and other criteria. Each queue has its own scheduling algorithm.

  Multilevel Feedback Queue Scheduling : Similar to Multilevel Queue Scheduling but now the processes are allowed to move among various queues

3.7 TERMINAL QUESTIONS

1. Discuss the key differences between Multilevel Queue Scheduling and Multilevel Feedback Queue Scheduling.

2. If the quantum time of round robin algorithm is very large, then it is equivalent to scheduling algorithm?

3. Consider the 3 processes, P1, P2 and P3 shown in the table.

Process	Arrival time	Burst Time
P1	0	5
P2	1	7

P3	3	4

The completion order of the 3 processes under the policies FCFS and RR2 (round robin scheduling with CPU quantum of 2 time units)

4. Which scheduling algorithms may lead to starvation and why?

5. Consider the following set of processes, with the arrival times and the CPU-burst times given in milliseconds.

Process	Arrival Time	Burst Time
P1	0	5
P2	1	3
P3	2	3
P4	4	1

What is the average turnaround time for these processes with the preemptive shortest remaining processing time first (SRPT) algorithm?

6. For the processes listed in the following table, which of the following scheduling schemes will give the lowest average turnaround time?

Process	Arrival Time	Processing Time
A	0	3
B	1	6
C	4	4
D	6	2

a) First Come First Serve b) Non – preemptive Shortest Job First c) Shortest Remaining Time d) Round Robin with Quantum value two.

7. What are the disadvantages of FCFS, SJF and round robin scheduling algorithms?

8. How does starvation possible in preemptive SJF algorithm. What is its solution?
9. What are the differences between multilevel queue and multilevel feedback queue scheduling algorithm?
10. How does the round robin scheduling is better than FCFS and SJF scheduling?

UNIT-IV CONCURRENT PROCESS

Structure

4.1 INTRODUCTION

Processes are designed for two purposes: first is to allow multiple processes to work together for a single task and second is to allow more than one process to execute parallely and communicate to each other in order to access common resources.

Thus, when processes compete for each other and cooperate on a task, they must communicate to each other to share data. Any process that share data with other processes is called as cooperating process. Otherwise it is an independent process. A pattern of communication among cooperating processes to access shared data or resources is known as interprocess communication. However, concurrent accesses by multiple cooperating processes to a shared data cause data inconsistency. In this chapter, we will discuss orderly execution of concurrent processes that share common data or resources to ensure data consistency. More precisely, we will discuss mutual exclusion way of interprocess communication.

4.2 OBJECTIVES

After studying this chapter, you should be able to:

- Understand the basic concepts of concurrency.

- Discuss the critical-section problem, whose solutions can be used to ensure the consistency of shared data.
- Understand the race conditions and mutual exclusion requirements.
- Understand hardware approaches to support mutual exclusion.
- Explain semaphores and monitors.

4.3 RACE CONDITIONS

When multiple cooperating processes try to access common shared data for reading and writing, this causes several issues. To illustrate this, let us consider a common example of a print-spooler. Whenever any process want to print a file, it adds the file name in a special spooler directory. Another process called printer daemon periodically checks the spooler directory. If it finds any file name in the directory, it prints them immediately and remove the file entry from the directory. The spooler directory contains a large number of slots which are numbered as 0,1,2… and each slot is capable of holding a file name. Consider that there are two variables: out and in. The 'out' variable holds a pointer to next file to be pointed. While, 'in ' variable holds a pointer to next free slot in the spooler directory. These two variables are present in a file which is available for all processes. At some point of time, imagine that slots 0 to 3 are emptied because associated files are printed. While, slots 4 to 6 are full because associated files are queued for printing. Consider that there are two process A and B, each one is queueing files for printing as shown in figure 4.1.

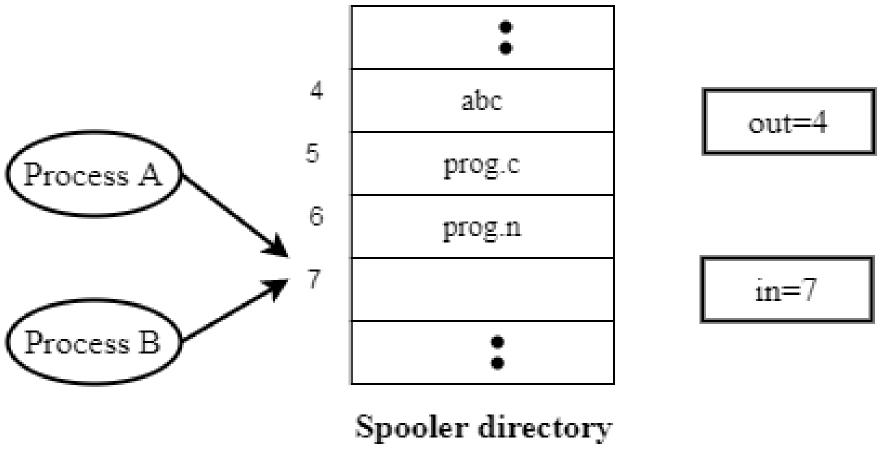


Figure 4.1: Two processes want to access shared memory at the same time [3].

The process A reads the 'in' variable and stores 7 in its local variable next_free_slot. Other process B also reads the 'in' variable and stores 7 in its local variable next_free_slot. Now, both process think that the next free slot is 7. The process B continue to run and reads the 'in' variable. It stores a new file name to be printed in slot 7 and update the 'in' variable as 8. Eventually, the process B gets interrupted and the process A continue to run again where it left off last time. The process A reads its variable next_free_slot which is 7. It stores the new file name to be printed in slot 7 which replaces the file name that process B just put there. After that, it updates the 'in' variable as 8. The process B hopes for the output for ever that will never come. The situation like this where multiple processes running concurrently try to manipulate a common shared data and their outcome depend on particular order in which the accesses takes place is called as race condition. The race condition can be avoided by ensuring only one process at a time to access a shared data. This guarantee can be achieved by synchronizing these processes in some way. This unit concerns with various ways of achieving process synchronization in cooperating processes.

4.4 SHARED DATA AND CRITICAL SECTION

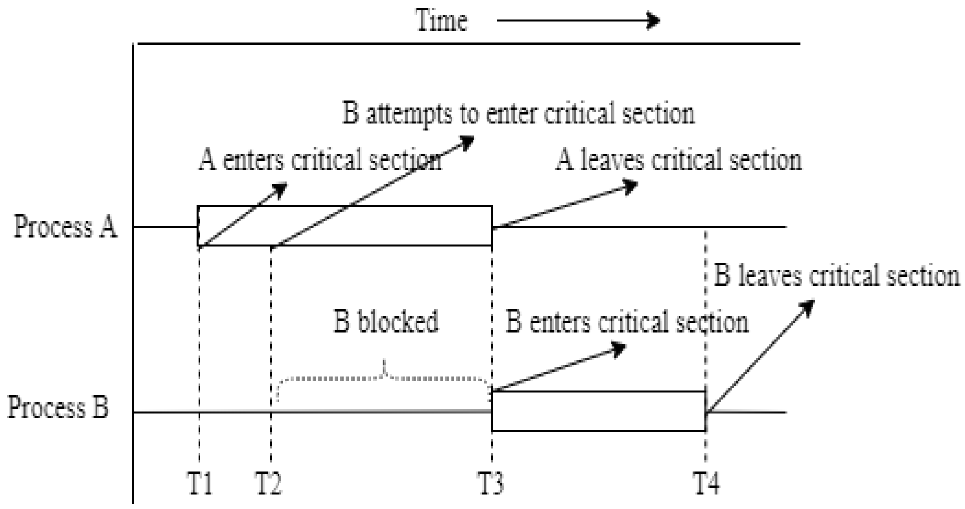
In previous section, we have seen the race condition which happens because the process B starts using one of the shared variable before the process A has finished using it. The key to avoid this problem is to only allow one process at a time to read/write a shared variable or file while, preventing other process to do the same thing. A race condition do not occur during internal computation and other things by a process. But, when the process accesses a shared data for other important things, a race condition may arise. The part of the program of a process where access to shared data take place is called as critical section or critical region. A race condition can be avoided by allowing only one cooperating process to execute its critical section at a time, while preventing other cooperating processes to execute their critical section.

For Example, consider a system consisting of n processes {P0, P1... Pn} implemented a protocol to avoid race condition. A general structure of any process Pi implemented the protocol is shown in figure 4.2.

```
do {

    entry section

        critical section

    exit section

        remainder section

} while (TRUE);
```

Figure 4.2 : General structure of a typical process Pi[1]

Each process contains a section of code called as critical section where the process changes the shared variables or a file. The entry section in the code of a process controls entry to its critical section. Once the process enters into its critical section, the other processes critical section are locked and they cannot enter into their critical section. Just after the critical section, there is an exit section which releases the lock to enter into their critical section. This indicates other interested process that it can now enter into its critical section. The rest of code other than critical section, entry section and exit section is known as remainder section. The above execution sequences is illustrated in figure 4.3.



Time
B attempts to enter critical section
A enters critical section
A leaves critical section
Process A
B leaves critical section
B blocked
B enters critical section
Process B
T1
T2
T3
T4

Figure 4.3 : mutual exclusion in critical section [3].

Here at time T1, a process A enters into its critical section. After some time, at time T2, another process B tries to enter into its critical section. But it is stopped to enter its critical section until time T3 because the process A is already in its critical section. At time T3 when the process A exits from its critical section, the process B enters into its critical section. At time T4 the process B also exits from its critical section and now no processes are in their critical section.

A critical section problem is to design a protocol for cooperating processes to ensure the execution sequence governed by process structure shown in figure 4.2. In general, any solution to the critical section problem must satisfy the following three conditions:

1. Mutual Exclusion : Only one process at a time can be executing in their critical section.
2. Progress : No process running outside its critical region may block other interested processes to enter their critical section.
3. Bounded Waiting : No process should wait forever to enter its critical region.

Illustrative question : The following two functions P1 and P2 that share a variable B with an initial value of 2 execute concurrently.

```
P1()
{
  C = B – 1;
  B = 2*C;
}
P2()
{
  D = 2 * B;
  B = D - 1;
}
```

What is the number of distinct values that B can possibly take after the execution?

Solution : Both P1 and P2 process can concurrently executes their statements in following ways:

First possibility :

```
 C = B – 1;   // C = 1
 B = 2*C;    // B = 2
 D = 2 * B;   // D  = 4
 B = D - 1;   // B  = 3
```

Second possibility :

```
 C = B – 1;   // C = 1
 D = 2 * B;   // D = 4
 B = D - 1;   // B = 3
 B = 2*C;    // B = 2
```

Third possibility :

```
 C = B – 1;   // C = 1
 D = 2 * B;  // D =  4
 B = 2*C;   // B = 2
 B = D - 1;  // B = 3
```

Forth possibility :

```
D = 2 * B;  // D =  4
C = B – 1;   // C = 1
B = 2*C;   // B = 2
B = D - 1;   // B = 3
```

Fifth possibility :

```
D = 2 * B;  // D =  4
B = D - 1;   // B = 3
C = B – 1;   // C = 2
B = 2*C;  // B = 4
```

We can notice that there are 3 different values of B: 2, 3 and 4.

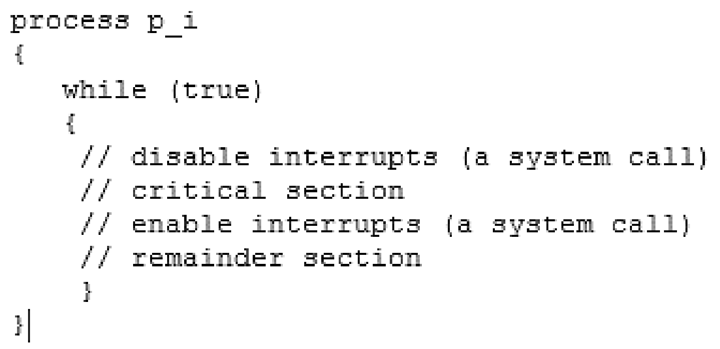
Check your progress

1. A critical section is a program segment
   a) Which should run in a certain specified amount of time
   b) Which avoids deadlocks
   c) Where shared resources are accessed
   d) Which must be enclosed by a pair of semaphore operations, P and V
2. What do you mean by race condition?
3. What do you mean by critical section?
4. What are the conditions that must be satisfied by any solution to a critical section problem?

4.5 SOLUTIONS TO CRITICAL SECTION PROBLEM

In this section, we will discuss various solutions to critical section problem. The key idea in these solutions is that when a process is busy in updating shared memory in its critical section, no other process could enter their critical section.

Disabling Interrupts (Hardware Solution) : This is a simplest solution which disable all interrupts when a process enters its critical section and enables all the interrupts just after leaving the critical section. A general structure of processes under this scheme is shown in Figure 4.4.



```
process p_i
{
   while (true)
   {
    // disable interrupts (a system call)
    // critical section
    // enable interrupts (a system call)
    // remainder section
    }
}
```

Figure 4.4 : General structure of a typical process

Once a process disable all the interrupts, the CPU cannot switch to other processes when the shared memory is updated. This restricts other processes to enter in their critical section. Disabling interrupts is a good technique which is useful within an Operating system, but it cannot be used for user processes.

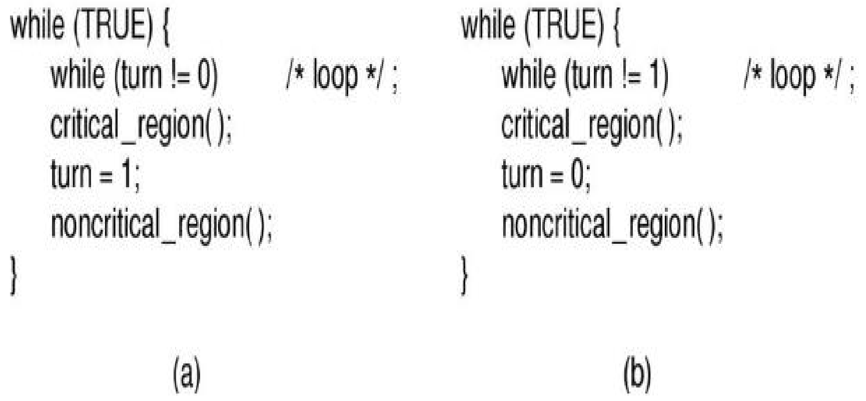
Disadvantages :

1. It is not a good decision to give user processes all the power to turn off interrupts. This is because, if a user process turns off all interrupts and it never turn on again, then the other processes will never get CPU for their execution.
2. This solution is not feasible in a multiprocessor system because when a process enters into its critical section and turns off all interrupts, a message should be passed to all processors about the disabling of interrupts which will cause delay in entering into its critical section.

Lock Variables (software solution) : This solution uses a single, shared lock variable which is initially set to 0. When a process wants to enter into its critical section, it first checks the lock variable. If the lock variable is 0, the process enters into its critical section. Otherwise, the process waits until the lock variable become 0. Thus, a 0 value of the lock indicates that there is no processes in their critical section. A value of 1 of the lock variable means one of the processes is into its critical section.

Drawback : This solution suffers from exactly same flaw that we saw in spooler directory case. To illustrate this, suppose a process A sees the lock variable is 0. Before it sets the lock variable to 1, another process B is scheduled to run and it sets the lock variable to 1 and enters into its critical section. When the process A runs again, since it already read the lock variable with 0 value, it also sets the lock variable to 1. Now, there are two processes simultaneously into their critical section.

Strict Alternation (software solution) : In this solution, an integer variable turn is used to allow which process to enter in its critical section. For example, if turn is 0, process A can enter into its critical section otherwise process B can enter into its critical section. The structure of two processes using this solution is shown in figure 4.5. Initially a process A checks a turn variable and finds that it is 0, so it enters into its critical section. At the same time, process B tries to enter into its critical section but it finds the value of turn variable is 0, so it waits. Process B continuously checks turn variable for 1 value in a tight loop. This continuous checking of a lock value till some other value appears is called as busy waiting and the lock being used is called as spin lock. Eventually, process A leaves its critical section and changes turn variable to 1. It quickly finishes its non-critical section and then goes to top of its while loop. Now both processes are outside their critical section. When process A again tries to enter its critical section, it is stopped because the turn variable is 1 in its while loop. But process B finds turn variable to be 1 in its while loop, so it enters into its critical section. When it is done with its critical section, it changes turn variable to 0 and exit from its critical section. Now, if process B again wants to enter its critical section, it cannot enter because turn variable is 0 in its while loop.



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

(a)

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

(b)

Figure 4.5 : A proposed solution to the critical region problem. (a) code for Process A (b) code for Process B

Disadvantage : This solution violate progress (condition 2) requirement of solution to critical section problem because process B is blocked to enter into its critical section, even other process A is outside of its critical section. This solution allows processes strictly in an alternate way to enter their critical section. However the solution ensures a race condition will never happen.

Peterson's Solution (software solution) : Peterson's solution is a software based classical solution to critical section problem which work for only two processes. Unfortunately, this solution does not work on modern computer system due to machine language instructions like load and store. Peterson's solution requires two shared data items:

int turn;

Boolean flag [2];

The turn variable is used to indicate which process's turn it is to enter its critical section. For example, if turn==i, then process Pi is allowed to enter its critical section. The boolean variable flag is used to indicate desire of a process to enter its critical section. For example, if flag[i] is true, then process Pi want to enter its critical section. A general structure of any process in Peterson's solution is shown in figure 4.6.The while loop without body followed by ; is busy loop which terminate the loop when the loop condition is true. The loop causes the process Pi to wait as long as process Pj has the turn and it want to enter its critical section. Just after its critical section, process Pi reset the flag variable to false so that process Pj can enter its critical section, if it is waiting. We can show that this solution satisfy all the requirements of solution to critical section problem.

1. Mutual exclusion is preserved:
2. The progress is satisfied:
3. Bounded waiting requirement is met:

```
do {

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

      critical section

    flag[i] = FALSE;

      remainder section

} while (TRUE);
```

Figure 4.6 : The structure of process Pi, in Peterson's solution

In the above diagram, the entry and exit sections are enclosed in boxes. In the entry section, process Pi first raise flag[i] to true indicating a desire to enter the critical section. Then turn is set to j to allow the process Pj to enter its critical section if it desires. The variable turn indicates whose turn it is to enter its critical section. The value of turn variable can be either 0 or 1. i.e., if turn = = i, then process Pi is allowed to execute in its critical section. The while loop is for a busy waiting (notice the semicolon at the end), which makes process Pi wait as long as process Pj has the turn and process Pj wants to enter its critical section. In the exit section, process Pi lowers the flag[i] to false, allowing process j to continue if it has been waiting.
This solution satisfy all the requirements of any solution to critical section problem:

1. Mutual exclusion is preserved : Any process say Pi can enter its critical-section only if its while statement is false. In its while statement, if other process Pj is interested (i.e flag[j]= true) and it is Pj turn to enter its critical section (i.e turn= j), then the Pi will wait because Pj is already in its critical section. Even if, both processes are executing their code simultaneously and reached to

their while statement, the last process which will execute turn==j statement will enter into its critical section. As the result, only one process at a time can enter its critical section.

2. Process requirement is satisfied: Any process Pi can only be prevented to enter its critical section, if other process Pj is already in its critical section (i.e in Pi's while statement, flag [j]== true and turn== j). As soon as the process Pj exits from its critical section, it changes flag to false. At that time, the process Pi while statement becomes false and it enters its critical section.
3. In point 2 of progress requirement, we saw that the process Pi will wait for almost after one entry by Pj to its critical section. Soon after that, the flag changes to false and process Pi them can enter its critical section.

Illustrative question : Two processes P1 and P2, need to access a critical section of code. Consider the following synchronization construct used by the processes as shown below. Here, wants1 and wants2 are shared variables, which are initialized to false. Which one of the following statements is TRUE about the above construct?

(A) It does not ensure mutual exclusion.
(B) It does not ensure bounded waiting.
(C) It requires that processes enter the critical section in strict alternation.
(D) It does not prevent deadlocks, but ensures mutual exclusion.

```
 /* P1 */
while (true) {
 wants1 = true;
 while (wants2 == true);
 /* Critical
   Section */
 wants1=false;
}
/* Remainder section */
/* P2 */
while (true) {
 wants2 = true;
 while (wants1==true);
 /* Critical
   Section */
 wants2 = false;
}
/* Remainder section */
```

Solution :

Deadlock : Assume processes P1 and P2 execute their code simultaneously. Imagine both processes now execute their while loop concurrently. At this point both wants1 and wants2 are true. They will enter their critical section only when their while statement become false. But, the wants1 and wants2 will never become false because either of the

process cannot proceed further. So both processes P1 and P2 stuck into their while loop in busy waiting. This will cause a deadlock in the system.
Mutual exclusion : we have already seen that both processes cannot enter into their critical section together. Now, assume that process P1 already entered into its critical section which means wants1=true. Now when P2 also try to enter into its critical section, it stuck into its while loop in busy waiting since want 1 is still true. This ensures P2 cannot enter its critical section simultaneously with P1 and vice versa. So, this satisfies mutual exclusion property.
Bounded waiting : Consider the same situation as described in mutual exclusion where process P2 stuck into its busy waiting. So P2 already requested to enter into its critical section. As soon as process P1 exits from its critical section, the wants1 becomes false. Now, P2 process can immediately enter its critical section. So, bounded waiting condition is also satisfied because P1 process is restricted to access its critical section only one time after P2 has requested to enter its critical section.
So, option D is correct.

Check your progress

1. What is the meaning of the term busy waiting? Can busy waiting be avoided altogether? Explain your answer.
2. What is the drawback of strict alternation solution to critical section?
3. What is the major shortcoming of lock variable based solution to critical section?

4.6 SEMAPHORE

All the algorithms discussed above suffers from a major problem of busy waiting which wastes much of CPU time. It does not seem wise for a process to spend entire time quantum in busy waiting for a shared variable to change its value. This has inspired the development of a solution based on semaphore.

Basic overview : semaphore is an integer variable with two atomic operations: wait() and signal(). Below are the implementation of these two operations.

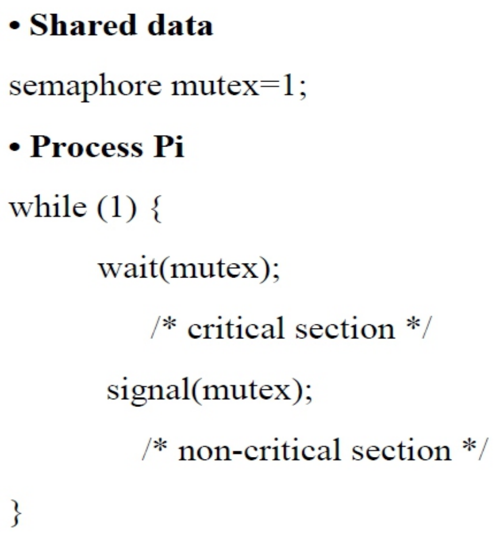
```
wait(S):
    while (S <= 0); /* wait */
    S--;
signal(S):
      S++;
```

Processes can change the semaphore value only through wait () and signal() operations. Only one process at a time is allowed to change a semaphore value i.e when a process modifies a semaphore value, no other processes is allowed to simultaneous modify its value. Wait() and signal() operations are sometimes also called as P and V respectively.

The semaphore can be either Counting Semaphore or Binary Semaphore. The value of a counting semaphore can be any integer, while the value of a binary semaphore can be either 0 or 1. The binary semaphore is sometimes also called as mutex lock because it provides mutual exclusion.

Binary semaphore usage :

1. The binary semaphore can be used to solve critical section problem involving n processes. The general structure of program implementing this solution is shown in figure 4.7.



• **Shared data**

```
semaphore mutex=1;
```

• **Process Pi**

```
while (1) {
        wait(mutex);
             /* critical section */
         signal(mutex);
             /* non-critical section */
}
```

Figure : 4.7 Binary semaphore based solution to critical section problem

The binary semaphore mutex variable is initiated to 1. This implementation of semaphore provides mutual exclusion and satisfy progress. However, depending on the different implementation of semaphores, it may or may not provide bounded waiting.

2. We can also use binary semaphore to solve synchronization problems. For example, consider a system with two concurrent process P1 and P2. Assume that process P1 contains S1 statement and P2 contains S2 statement. We have to ensure that, process P2 only executes after execution of process P1. This can be achieved by using a binary semaphore variable synch which is initialized to 0. The code structure of process P1 and P2 should be as follow:

Code structure of process P1:

```
S1;
signal(synch);
```

Code structure of process P2 :

wait (synch);

S2;

Counting semaphore usage : Counting semaphore can be used to grant access to a finite number of resources (for example, 10 printers). This can be achieved by initializing a Counting semaphore variable count to the number of resource. Whenever any process want to use the resource, it performs wait () operation on the count variable which decrements its value. And whenever, any process releases the resource, it performs signal() operation which increments the count value. When the count variable value becomes 0, this means that all the resources has been used up. After that when any process wants to use the resource, it must wait until the count value becomes greater than 0 i.e. some other process has performed signal () operation.

Avoiding busy waiting : The main drawback of the semaphore discussed above is busy waiting. In the multiprogramming system, where one process is busy in its critical section, while other process wish to enter its critical section will continuously loop in its entry code. This causes wastage of CPU cycles which could be used productively for other processes. The problem of busy waiting can be overcome by modifying the definition of wait () and signal() operations as shown below in figure 4.8:

```
• wait(S):
        S.value--;
        if (S.value < 0) {
            add this process to S.L;
            block;
}
• signal(S):
         S.value++;
         if (S.value <= 0) {
             remove a process P from S.L;
             wakeup(P);
}
```

Figure : 4.8 Modified wait () and signal () operations to avoid busy waiting

Now when a process executes the wait () operation and finds that semaphore value is not positive, the process will call a block () system call to move to waiting state instead of doing busy waiting. Each semaphores variable is associated with a list of waiting processes. The block () system call moves the process to a waiting queue associated with the semaphore variable S. The CPU scheduler now schedules another process to execute.

The blocked process waiting on semaphore S will be restarted when some other process executes signal() operation. The signal () operation calls a wakeup() system call which moves the blocked process to ready queue and changes its state to ready state. A negative value of semaphore variable represents number of processes waiting for a semaphore variable. The wait () and signal() system call are atomic that cannot execute simultaneously on the same semaphore variable.

Deadlock and starvation: A deadlock situation is possible in the solution based upon wait () and signal() operation discussed above. To illustrate this, consider two processes P0 and P1 with two semaphore variable S and Q, initially both set to 1 as shown below.

```
      P0              P1
   wait(Q);        wait(R);
   wait(R);        wait(Q);
     ...             ...
  signal(R);      signal(Q);
  signal(Q);      signal(R);
     ...             ...
```

Imagine that process P0 executes wait (S) and process P1 executes wait (Q) operation. Now, when process P0 executes wait (Q) operation, it must wait until P1 executes signal(Q) operation. Similarly, when P1 executes wait (S), or must wait until process P0 executes signal (S) operation. But the signal () operation cannot be executed because both process cannot proceed its further execution. This leads to a deadlock situation. A situation where processes wait indefinitely within the semaphore is called starvation.

Check your progress

1. What is the difference between binary semaphore and counting semaphore?
2. Give one usage of each of following :
   a) Binary semaphore
   b) Counting semaphore

4.7 CLASSICAL PROBLEM OF SYNCHRONIZATION

In this section, we will discuss how semaphores can be used to solve some classical synchronization problems. These problems are used to test any newly proposed synchronization scheme.

Bounded buffer using semaphores :

A Bounded Buffer problem is generalization of producer and consumer problem where a shared buffer of n slots is accessed in a mutual exclusion way. You can also think this as a producer which is producing full buffer and a consumer as producing an empty buffer. In this semaphore based solution, two counting semaphore variables are used: full and empty. The full variable is initialized to 0 which keep track of currently occupied slots in the buffer. The empty variable is initialized to N which keeps track of currently available slots in the buffer. The figure 4.9 shows the general code structure for any producer and consumer process.

```
• Shared data:
   semaphore fullslots, emptyslots, mutex;
   full=0; empty=n; mutex=1;
• Producer process:
   while (1) {
       produce item;
       wait(emptyslots);
       wait(mutex);
       add item to buffer;
       signal(mutex);
       signal(fullslots);
   }
• Consumer process:
   while (1) {
       wait(fullslots);
       wait(mutex);
       remove item from buffer;
       signal(mutex);
       signal(emptyslots);
       consume item;
   }
```

Figure 4.9 : The code structure of producer and consumer process

The Readers-Writers Problem

In this problem, there are multiple reader processes and multiple writer processes, each try to access a shared data or dataset, for example a railway reservation dataset. The reader processes can access a shared data concurrently with writer processes. But a writer process can access the shared data in a mutual exclusive way. The goal here is too maximize concurrency while avoiding data inconsistency. A reader process can see the shared data or dataset but cannot make any changes in it just like a person watching available seats during railway reservation. A writer process can change the shared data. For example, a person is booking a seat in railway reservation. But we must ensure that two writer process cannot simultaneously access the shared data or dataset for example, two person cannot book a same available seat (remember counter++ and counter--!). However, multiple readers are allowed to view the shared data as long as no writer is updating the shared data. In other words, when a writer process already gets permission to access the shared data, no readers are allowed to read the shared data. For example we can allow multiple customers to view the available seats just to avoid delay. A general code structure for a reader and writer process are shown in figure 4.10.

- **Shared data:**

```
semaphore wrt, mutex;
int readcount;
wrt=1; mutex=1; readcount=0;
```

- **Writer process:**

```
while (1) {
    wait(wrt);
        /* perform writing */
    signal(wrt);
}
```

- **Reader process:**

```
while (1) {
    wait(mutex);
    readcount++;
    if (readcount == 1) wait(wrt);
    signal(mutex);
        /* perform reading */
    wait(mutex);
    readcount--;
    if (readcount == 0) signal(wrt);
    signal(mutex);
}
```

Figure 4.10 : The code structure of a reader and writer process

In the code, a readcount variable is initialized to 0 which keeps track of number of readers processes reading the shared data. The semaphore mutex is initialized to 1 which ensures mutual exclusion when the readcount variable is updating. The semaphore wrt is initialized to 1 which allows multiple reader to access the shared variable in a mutual exclusive way.

Note that in this solution a reader may wait(wrt) while inside mutual exclusion of mutex. Is this OK? This solution based on semaphore is a reader-preference solution where writers may starve!. We don't want any reader to keep waiting unless a writer is in critical section.

The Dining-Philosophers Problem

Dining Philosophers problem is another classical synchronization problem for allocating a set of limited resources among a group of processes in a deadlock free and starvation free way. In this problem there are 5 philosophers sitting around a round table with a bowl of endless rice. The table is laid with 5 chopsticks as shown in figure 4.11.

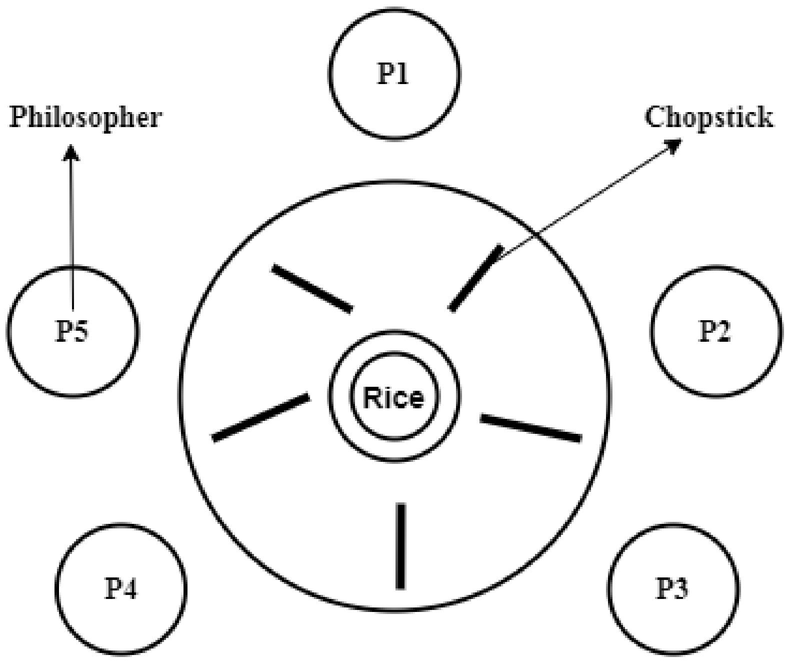


Figure 4.11: The situation of the dining philosophers [1].

Each philosopher spends their time in interacting, thinking and eating. When a philosopher feels hungry, he picks up one chopstick from his left and another from his right and starts eating. When a philosopher is finished with eating, he puts down both the chopsticks to its original place and starts thinking again. A solution to this problem based on semaphore is shown in figure 4.12.

- Shared data:

```
semaphore chopstick[5];
chopstick[0..4]=1;
```

- Philosopher i:

```
while (1) {
  wait(chopstick[i]);
  wait(chopstick[(i+1)%5]);

   /* eat */

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

   /* think */

}
```

Figure 4.12 : The structure of philosopher i.

In the code structure, a set of 5 chopsticks is shown with a semaphore variable chopsticks [5]. Each hungry philosopher first picks his left chopstick chopstick[i] and then his right chopstick chopstick[(i+1)%5]. But, the main problem will arise when all philosopher get hungry at same time. In this case, each philosopher first picks their left chopstick. The time when they start picking their right chopstick, it becomes unavailable and they all will wait forever. This lead to a deadlock situation.

Some possible solution to the deadlock problem are:

1. Allow only 4 philosophers for eating at the same time.
2. Allow any hungry philosopher to pick up chopsticks only when both are available.

Allow odd philosophers to pick up their left chopsticks first and even philosophers to pick up their right chopsticks first.

4.8 MONITOR

Semaphores can solve various synchronization problems very effectively. But it should be used very carefully because even a single improper use by a process can break an implementation. Monitor which is a high level programming language construct in another way for achieving synchronization of process. Similar to C++, Java, Monitor provides abstract data type, shared variables, and methods to implement process synchronization. A monitor is essentially a class with following properties:

1. It is the collection of four components: initialization, private data, monitor procedures, and monitor entry queue bind into a single class as shown in Figure 4.13.
2. The initialization component contains the code that is used exactly once when the monitor is created.
3. The processes running outside the monitor can’t access the internal variable of monitor. The private data are not visible from outside of the monitor.
4. The monitor procedures can be called from outside of the monitor.
5. Only one process at a time can execute code inside monitors.
6. The monitor entry queue contains all threads that call monitor procedures but have not been granted permissions.

```
monitor monitor_name
{
    shared variables;
    procedure P1(...) {
    ...
    }
    procedure P2(...) {
    ...
    }
    ...
    initialization/constructor;
}
```

Figure 4.13 : Syntax of a monitor

Monitor consists of one or more data type which is called as condition. Any variable of condition type supports only two operations: wait and signal. For example, if X is a condition variable, then it can perform the legal operations: X.wait( ) and X.signal( ). The wait operation blocks one process and adds it to a list associated with that condition variable. The signal operation wakes up exactly one process from the condition's list of waiting processes. For illustration, figure 4.14 shows a monitor with an entry queue of processes that are waiting for their turn to execute monitor methods/procedure.

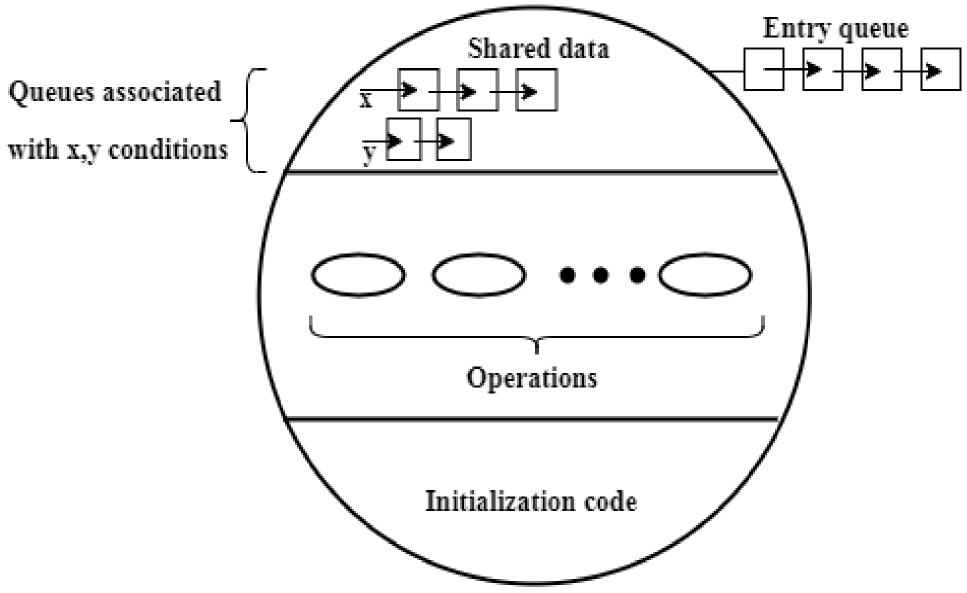


Figure 4.11: The situation of the dining philosophers [1].

There is a potential problem with this method. If any process P within the monitor issues a signal operation, then it would wake up any process Q that would also within the monitor. This cause two processes running simultaneously within the monitor which violates the exclusion requirement. There are two possible solutions to this problem:

- Signal and wait – If any process P executes a signal operation which wakes up another process Q, then P should wait either for Q to leave the monitor or for some other condition.
- Signal and continue - When P executes the signal operation, Q should wait either for P to exit the monitor or for some other condition.

Concurrent Pascal offers a third alternative where signal call causes the signaling process to immediately exit the monitor, so that the waiting process can then wake up and proceed. Many programming languages including concurrent Pascal, Mesa, C# (pronounced C-sharp), and Java have already incorporated the idea of the monitor as described in this section.

Dining-Philosophers Solution Using Monitors :

Now we will discuss a solution to the dining philosophers' problem based on monitors. In this solution, we impose a restriction that a philosopher may pick up chopsticks only when both are available. There are two key data structures used in this solution:

1. enum { THINKING, HUNGRY,EATING } state[5]

   By using this data structure, a philosopher can be set to eating state only when neither of his/her adjacent neighbours are eating. This can be ensure by checking this condition: state[(i + 1) % 5] != EATING && state[(i + 4) % 5] != EATING .

2. condition self[5]

   This condition is used to delay any hungry philosopher who is unable to acquire chopsticks.

```
monitor DiningPhilosophers
  {
        enum { THINKING; HUNGRY, EATING) state [5] ;
        condition self [5];
        void pickup (int i) {
              state[i] = HUNGRY;
              test(i);
              if (state[i] != EATING) self [i].wait;
        }
     void putdown (int i) {
              state[i] = THINKING;
              // test left and right neighbors
               test((i + 4) % 5);
               test((i + 1) % 5);
      }

     void test (int i) {
                     if ( (state[(i + 4) % 5] != EATING) &&
                     (state[i] == HUNGRY) &&
                     (state[(i + 1) % 5] != EATING) ) {
                          state[i] = EATING ;
                            self[i].signal () ;
                      }
              }
           initialization_code() {
                   for (int i = 0; i < 5; i++)
                   state[i] = THINKING;
              }
     }
```

Figure : 4.15 Dining-Philosophers Solution Using Monitors

The distribution of the chopsticks is controlled by the monitor DiningPhilosophers, whose definition is shown in Figure 4.15. Before start eating, each philosopher must invoke following sequence of operations.

1. DiningPhilosophers.pickup( ) - Acquires chopsticks, which may block the process.
2. eat
3. DiningPhilosophers.putdown( ) - Releases the chopsticks

This solution ensures that no two neighbours are eating simultaneously and thus no deadlocks will occur.

4.9 SUMMARY

In Summary,

- We saw that the processes must communicate with each other to share data. However, concurrent access to shared data by multiple processes may result in data inconsistency. So OS must provide mutual exclusion way of achieving process communication.
- We learnt that computer hardware provides several operations to ensure mutual exclusion. But for most developer these hardware-based solutions are too complicated for them to use. To overcome this problem, semaphores can be used efficiently to solve data inconsistency problem and provide mutual exclusion way of achieving process communication.
- We have discussed various synchronization problems such as the bounded-buffer problem, the readers-writers problem, and the dining-philosophers problem because they are examples of a large class of concurrency-control problems. Almost every newly proposed synchronization scheme is test using these classical problems.
- Lastly, we saw that Monitors are a high-level programming language construct with shared data and methods to achieve Process synchronization.

4.10 TERMINAL QUESTIONS

1. Explain why interrupts are not appropriate for implementing synchronization primitives in multiprocessor systems.
2. Illustrates how the Peterson solution satisfies all three requirements of any solution to critical section problem.
3. The Bounded buffer problem is also known as __________.

4. Explain three classical synchronization problem with their solutions.
5. What is the main disadvantage of semaphore based solution to critical section problem? How does it is removed?
6. How does deadlock is possible in semaphore based solution to critical section problem?
7. Explain strict alternation solution to a critical section problem.
8. How does the lock variable provide solution to critical section problem?
9. How do cooperating processes differ from independent processes?
10. Illustrates how race condition occurs.
11. What is a general solution to a race condition?

BIBLIOGRAPHY

1. Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. Operating System Concepts (8th. ed.) : Wiley Publishing, 2008
2. Crowley, Charles. Operating systems: a design-oriented approach. McGraw-Hill Professional, 1996.
3. Modern Operating Systems Second Edition by Andrew S. Tanenbaum Publisher: Prentice Hall Ptr
4. Stallings, William. Operating systems: internals and design principles. Boston: Prentice Hall, 2012.
5. Ritchie, O. M., and Ken Thompson. "The UNIX time-sharing system." The Bell System Technical Journal 57.6 (1978): 1905-1929.
6. Deitel, Harvey M., Paul J. Deitel, and David R. Choffnes. Operating systems. Pearson/Prentice Hall, 2004.

Uttar Pradesh Rajarshi Tandon
Open University

Master of Computer Applications

DCECS - 106

Operating System

BLOCK

2

Block-2 Memory Management and UNIX Case Study

Course Design Committee

Course Preparation Committee

Manoj Kumar Balwant **Author**

Asstt. Professor

School of Science, UPRTOU, Allahabad

Prof. Manu Pratap Singh **Editor**

Professor, Department of Computer Science and Engineering

Institute of Engineering & Technology (Khandari campus)

Dr B R Ambedkar University

Agra,Uttar Pradesh

UPRTOU, Prayagraj-2022
DCECS -106 Operating System
ISBN : 978-93-83328-97-0
Reprint -

Printed and Published by Pro. (Dr.) Arun Kumar Gupta Registrar, Uttar Pradesh Rajarshi Tandon Open University, 2021

Printed By – M/s K.C.Printing & Allied Works, Panchwati, Mathura - 281003

COURSE INTRODUCTION

This Block consists of 4 units: 5,6,7,8. In fifth unit, deadlock problem is explained along with different methods of avoiding and preventing the deadlock. This unit also covers the various methods to recover from the deadlock. In sixth unit, various memory management concepts are described. It includes paging, segmentation, virtual memory, demand paging, page-replacement algorithms, and thrashing. The seventh unit explains about the working of magnetic disk and data organization on disks. In this unit, different Disk Scheduling Algorithms are described and the concept of swap space in virtual memory system is explained. The last unit of this block explores the history of the UNIX operating system and the principles on which Linux is designed. This unit also explains process scheduling and inter-process communication in UNIX. This unit is providing the learning about how memory management and file systems are implemented in UNIX.

UNIT-V DEADLOCK

Structure

5.1 INTRODUCTION

In a multiprogramming environment, multiple processes execute concurrently and requires several resources from a finite set of resources. If an executing process needs a resource which is not currently available, it enters into waiting state. Sometime a waiting process never gets the resources because it may be held by another waiting process. This lead to a deadlock situation where processes wait for resources forever and none of them proceed their execution. For example, consider a system with two process A, B each try to print files on compact disk (CD). So the system has two resources: CD and printer. Imagine a sequence of operations where a process A first obtains ownership of the printer for printing a file. At the same time, process B obtains ownership on CD for reading a file. Now, process A tries to get ownership on CD but, it is told to wait for process B. Process B also tries to get ownership on printer but it is told to wait for process A. This situation leads to deadlock and neither of them can proceed further. In this chapter, we will discuss the deadlock problem in detail and various methods of handling the deadlock.

5.2 OBJECTIVES

After reading this unit

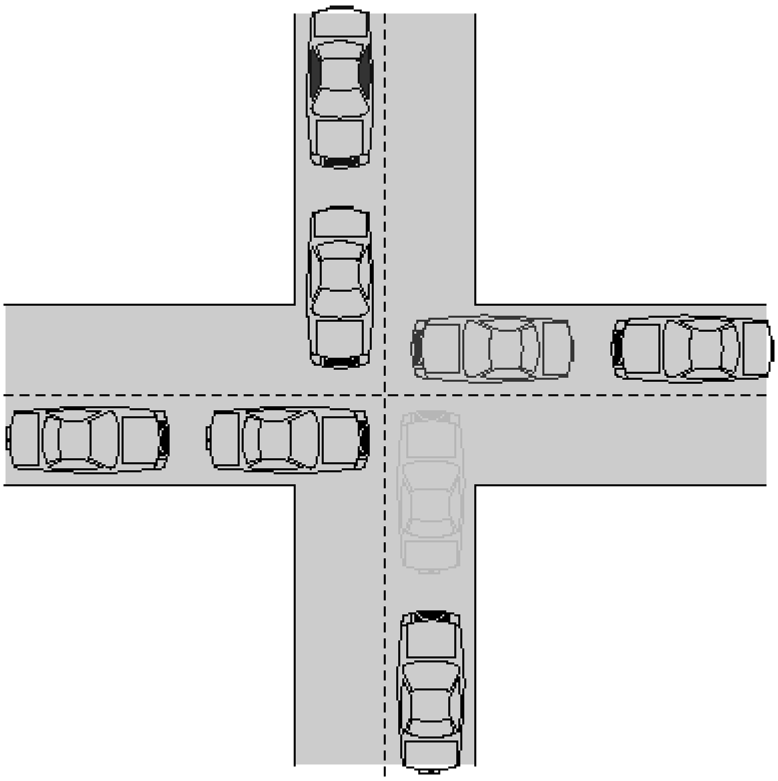
- You will able to explain deadlock problem.

- You will understand various ways to handle deadlock situation.

5.3 THE DEADLOCK PROBLEM

In Multiprogramming system, several processes compete for a finite set of resources. These processes enters in waiting state, if the resources are not currently available. Sometimes processes wait for ever because the resources they have requested is already held by theses waiting processes. This situation is called Deadlock.

Examples : In the automotive world a deadlocks happens when a situation arises as shown in Figure below. Here the cars can be thought of the processes and the spaces occupied by the cars can be thought of the resources.



Illustrative Example : A system contains three programs and each requires three tape units for its operation. What is the minimum number of tape units which the system must have such that deadlocks never arise?

Solution : If all three processes will hold 2 resources each and waiting for 1 more resource to complete its execution. Therefore, if there are 7 resources in the system, then at least one program must have 3 resources so that it will complete and free up all its resources which can be used by other program to complete their task and so deadlock can never occur. Thus we have:

Minimum number of resources= Number of Program * (maximum need-1) +1.

= 3*(3-1) + 1=7.

5.4 SYSTEM MODEL

For the sake of understanding deadlock problem, several assumptions and terminologies are made regarding a system. A system consists of a finite set of resources which are allocated to several processes as per their needs. For example, the resources can be printers, DVD, CPU cycles etc. These are physical resources. Resources can be logical also such as semaphores, locks, files and drivers. System resources can be divided into several classes or types and each class may have multiple identical resources (or instances of that resource). For example, when system has two CPUs then it has two instances of CPU type resource.

If a process requests for an instance of a resource type, then its request can be satisfied by allocating any one of the instances of that resource type. The process may request any number of resources but, it should not exceeds the total number of resources available in the system. Any process must request a resource before using it and must release the resource after using it. Thus a process uses resources in following order:

Request : The process must request the resource before using it. If the resource is currently available, it is granted immediately. Otherwise, it goes to waiting state and waits for the resource.

Use : The process utilizes the resource for its execution.

Release : After the resource has been used, it is released.

The request and release operations can be system calls. For example, allocate() and free memory(), open() and close() files, wait () and signal() operation. A deadlock may involve with either same resource types or different resource types. For example consider a system with one printer and one DVD drive. Suppose, a process pi is holding printer and another process pj is holding DVD drive. Now if, the process pi further requests for DVD drive and the process pj request for printer, then a deadlock will occur.

5.4.1 (NECESSARY) CONDITIONS FOR DEADLOCK

A deadlock can occur if following conditions hold simultaneously.

1. **Mutual exclusion :** Only one process at a time can use a resource. If another process requests the same resource, it must wait until the resource held by previous process is released.
2. **Hold and Wait :** A process is holding one resource and waiting for another resource which is held by another process.
3. **No Preemption :** Resources held by a process can only be released by the process itself after finishing their executions.
4. **Circular wait :** a sequence of waiting processes { p0,p1,p2…...pn} must exist such that process p0 is waiting for a

resource currently held by p1, p1 is waiting for a resource currently held by p2 and so on pn is waiting for a resource currently held by p0.

5.4.2 DEADLOCK MODELLING

In some cases the deadlock can be better understood by the resources-allocation graph. The resource-allocation graph can be described by the following properties.

1. A set of process {P1, P2, P3...Pn} represented as circles in the graph.
2. A set of resource categories {R1, R2, R3 ... Rn} represented as squares in the graph. The dot inside the resource indicates specific instances of the resource. (For example two dots may represent two instances of the resource type).
3. Request Edges – A directed arc from Pi to Rj indicates that process Pi has requested resource Rj, and is currently waiting for that resource to become available.
4. Assignment Edges – A directed arc from resource Ri to process Pj indicats that the resource Ri is currently held by the process Pj.

When the request is granted, a request edge can be converted into assignment edge by reversing the direction of the arc. The request edge is pointed toward the category box while assignment edge originates from particular dot inside the category box as shown in Figure 5.1

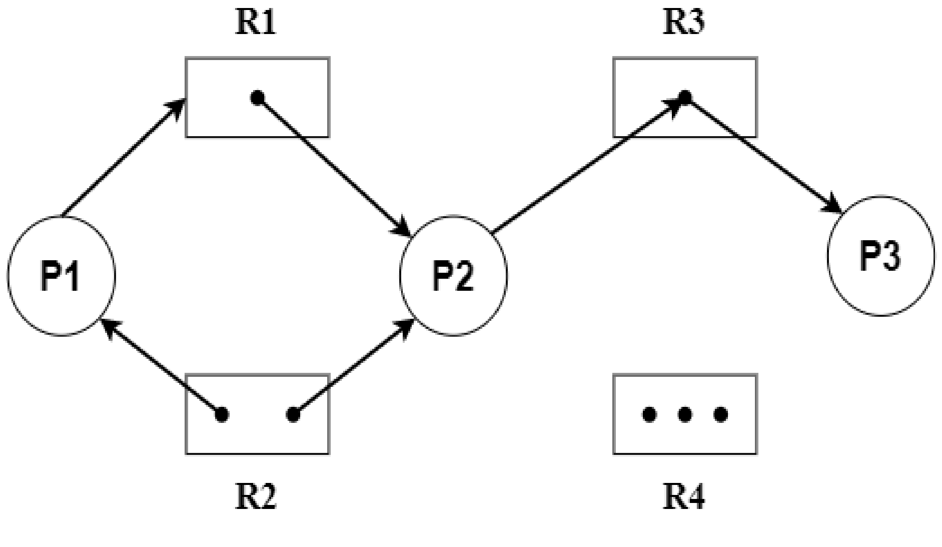


Figure 5.1 : Resource allocation graph [1].

If a resource allocation graph contains no cycles, then there is no deadlock present in the system. If the resource allocation graph contains cycles, and if there is single instances of each resource category then deadlock is present. If the resource allocation graph contains more than one instance of the resource categories, then the presence of cycle does not guarantee a deadlock. Thus, cycle in a multi instances resource allocation graph is a necessary condition but it is not a sufficient condition. For example, in Figure 5.2 the process P4 will release the resource type R2. Now, this resource can be used by P3 process to break the cycle.

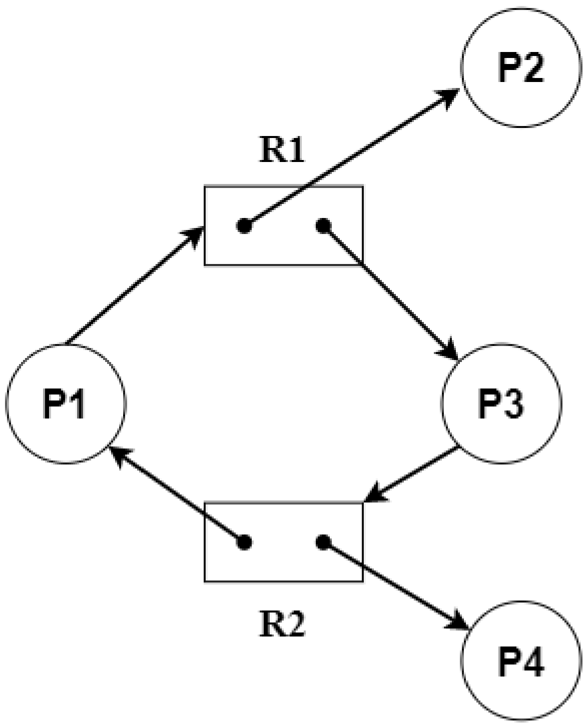


Figure 5.2 : Resource allocation graph with a cycle but no deadlock [1].

Check your progress

1. What does a cycle in a Resource Allocation Graph indicate?
2. Justify your answer : Whether a cycle in a resource allocation graph is sufficient condition for existence of deadlock in a system.
3. Describe necessary conditions for occurrence of a deadlock in a system.
4. Explain a deadlock with a real word example.

5.5 METHODS FOR HANDLING DEADLOCKS

We can deal with the deadlock in one of following ways

1. **Prevention or Avoidance :** In this approach, a protocol can be designed to ensure that a system will never enter into a deadlock state. This can be ensured by using either Deadlock Prevention or Deadlock Avoidance. Deadlock prevention consists of a set of methods to ensure that at least one of the necessary conditions of

deadlock cannot satisfy while allocating requests for resources. On other hand, Deadlock avoidance uses additional information in advance regarding a process requests to resources for its entire execution. This additional information includes available resources, allocated resources and future requests and release of each process. A decision for a current request of any process to be granted or delayed is made on the basis of this additional information. We will discuss Deadlock prevention and Deadlock avoidance in detail in next section.

2. **Detect and Recover Deadlock :** If the deadlock prevention or deadlock avoidance methods failed, then deadlock may occur in the system. In this method, we examine state of the system to detect whether a deadlock has happened or not. If there is deadlock in the system, we recover the system from deadlock states using the deadlock recovery methods.

3. **Ignore the deadlock :** In this method, if large number of processes enter in the deadlock state and more number of resources are held up for these waiting processes then system performances degrades. In the last, system stop working and we do not have any other option except to restart the system.

In general these methods are combined together to allocate resources to processes in an optimal manner to deal with the deadlock.

5.6 DEADLOCK PREVENTION

Deadlock can be prevented by not allowing the system any one of the four conditions to occur.

Avoid Mutual Exclusion : If we ensure that no resource will assigned exclusive to single process, then there will be no deadlock. Shared resources such as read-only files, read/write locks, memories etc. do not lead to deadlock. Unfortunately some resources like printers, tapes, CD-ROM Drive are not sharable and require exclusive access by the single process.

Avoid hold and wait : If we prevent a process that already holds resources to further request other resources, then we can prevent deadlock. This can be achieved by allowing all the processes to request for the resources before starting execution. So, if all the requested resources are available then they are allocated to processes, otherwise none of the resources will allocate to the process.

Attacking No Preemption : This can be achieved by ensuring that the scheduler should preempt a process, if it is holding some resources and requesting more resources that are not currently available. After pre-empting the process, resources become available for other processes. Alternatively, Scheduler should only schedule a process, if all its requested resources are available.

Avoid Circular wait : Circular wait can be avoided by numbering all the resources and strictly allowing the processes to request the resources in ascending order. So, if a process initially requested instances of resources type Ri, then another requests for instances of another resource type Rj can only be requested if Rj>Ri. For example, consider a process is holding resource number #11 and #15, it can only request resource number #16 or higher. In order to request a resource number #8, it must have to first release the resources number #11 and #15. Hence, with this rule the resource allocation graph will never construct the cycle.

5.7 DEADLOCK AVOIDANCE

The possible drawbacks with deadlock prevention algorithm are low device utilisation and low system throughputs. Deadlock avoidance is an alternative way of preventing deadlock in the system. This algorithm requires prior information such as number of available resources, number of allocated resources and maximum demands of each process. In this section we will discuss two deadlock avoidance algorithm. But, first we will discuss safe state and safe sequences which are important to understand these two algorithm.

Safe State : A safe state is a state where deadlock never occures. But, when a state is unsafe, a deadlock may or may not occur as shown in Figure 5.3. A system is said to be in safe state if, it could allocate the available resources to processes in some order such that the maximum needs of each process can be fulfilled without occurring a indefinite waiting. More formally, a safe state exists if, there is a safe sequence <p1,p2,p3……pn> such that, the resource requests of any process pi can be satisfied by the currently available resources plus resources held by its preceding pj process in the safe sequence with all j<i. For example, consider a system with 12 printers attached and the processes p0, p1, p2 are currently executing. The maximum needs and current allocation of each process at any time is shown below:

Process	Maximum needs	Current allocation
P0	10	5
P1	4	2
P2	9	2

The total available printers are 3=12-5-2-2.

The system is in safe state because there exists a safe sequence <p1, p0, p2>. The current needs of 2 printers (2=4-2) for p1 is satisfied because there are 3 available printers. When the need of p1 is fulfilled, it releases all the printers it holds (2 printers) and added to the currently available printers (5 printers). Next, needs of process p0 for 5 (5 = 10-5) printers satisfied with 5 (5=3+2) available printers. After satisfying the needs of p0, it releases its holding printers (5) and added to currently available printers (5). Finally, needs of p2 for 7 printers satisfied with 10 printers (10= 5+5) currently available printers.

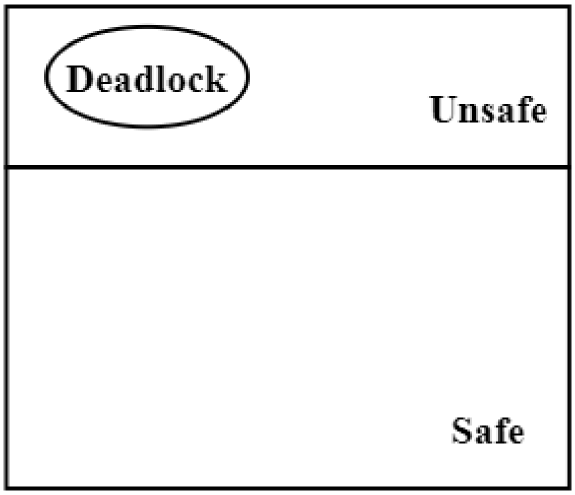


Figure 5.3 : Safe, unsafe and deadlock state [1].

Resource Allocation Graph Algorithm :

We can use the resource allocation graph with single instance of each resource discussed earlier for deadlock avoidance. But this time, in addition to the request and assignment edges, we will use claim edges for deadlock avoidance. A claim edge Pi-->Rj represents that the Pi process requests the Rj resource in near future. This edge is similar to request edge, but represented by dotted edge. The claim edge is converted to request edge when Pi process requests the Rj resource.

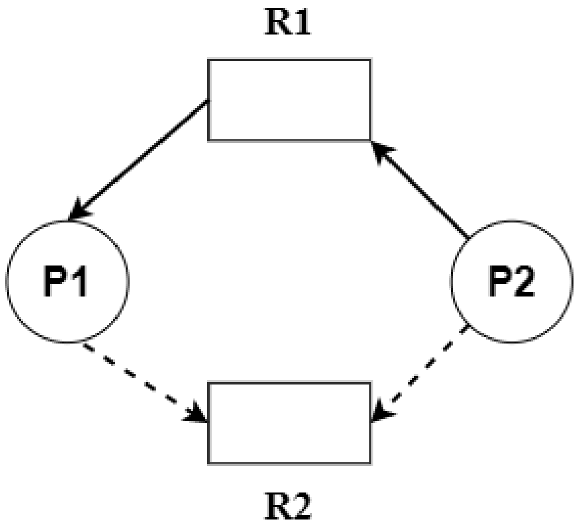


Figure 5.4 : Resource-allocation graph for deadlock avoidance [1].

Now consider a system as shown in Figure 5.4. Assume that the process P1 requests R2 resource. This request can be granted only if, converting the request edge P1---> R2 to an assignment edge R2--->P1 will not lead to formation of a cycle in the resource allocation graph. Because the cycle will indicate a deadlock in the system. If no cycle exists, the system is in safe state and the request can be granted immediately. But, if the cycle exists, the system is in unsafe state and the P1 process should wait for its request to be satisfied. This is also illustrated in Figure 5.5.

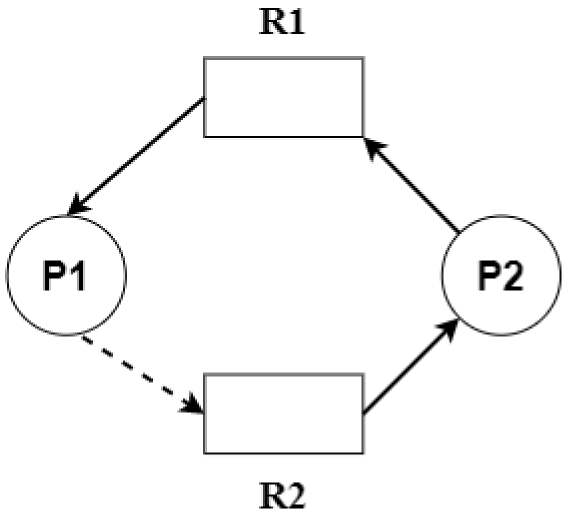


Figure 5.5 : An unsafe state in a resource-allocation graph [1].

Banker's Algorithm : The resource allocation graph algorithm which we have discussed in previous section does not applicable to a system having multiple instances of each resource type. The deadlock avoidance algorithm which is known as Banker's Algorithm is applicable to this type of the system. In this algorithm, a process prior to its execution, revels its maximum resources needed during execution. The algorithm checks this number of resources needed should not exceeds the total number of resources in the system. The algorithm grants a process access to a set of resources only if the allocation of these resources will lead the system to a safe state. Otherwise, these resources are not allocated to the process and it has to wait until the system gets enough resources when other processes release their resources.

Implementation of banker algorithm requires several data structures:

Available : It is a vector of length m which indicates number of instances of each resource type. For example, Available[j] equal to k represents k instances of resource type Rj.

Max : It is a matrix of size n*m which represents the maximum demands of each process. For example, Max[i][j] equal to k indicates the process Pi has k instances of resource type Rj.

Allocation : It is a matrix of size n*m which represents number of instances of each resource type currently allocated to each process. For example, Allocation[i][j] equal to k represents k instances of resource type Rj are currently allocated to a process Pi.

Need : It is also a matrix of size n*m which indicates the remaining need of each instances of each resource type for a process to finish its execution. For example, Need[i][j] equal to k represents there are k Instances of resource type Rj needed to process Pi to complete its execution.

Safety Algorithm :

This algorithm determines whether the current state of a system is safe or not, according to the following steps:

1. Let Work and Finish be vectors of length m and n respectively.
   a) Work is a working copy of the available resources, which will be modified during the analysis.
   b) Finish is a vector of booleans indicating whether a particular process can finish. (or has finished so far in the analysis.)
   c) Initialize Work to Available, and Finish to false for all process.
2. Find an i such that both (A) Finish[i] == false, and (B) Need[i] < Work. (This process has not finished, but could with the given available working set.) If no such i exists, go to step 4.
3. Set Work = Work + Allocation[i], and set Finish[i] to true. This corresponds to process i finishing up and releasing its resources back into the work pool. Then loop back to step 2.
4. If finish[i] == true for all i, then the state is a safe state, because a safe sequence has been found.

Illustrative Example 5.1 : Considering a system with five processes P0 through P4 and three resources types A, B, C. Resource type A has 10 instances, B has 5 instances and type C has 7 instances. Suppose at time t0 following snapshot of the system has been taken:

Process	Allocation	Max	Available
	A B C	A B C	A B C
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

Now, the need matrix and the safety sequence as:

Need = Max – Allocation

So, the content of Need Matrix is :

Process	Need		
	A	B	C
P_0	7	4	3
P_1	1	2	2
P_2	6	0	0
P_3	0	1	1
P_4	4	3	1

We can find a safe sequence by checking the safety condition for each process in order of their process IDs.

Step 1 : Checking of Safety condition for process P0
Need of process P0= need0 = (7, 4, 3)

If need0 ≤ Available => [(7, 4, 3) ≤ (3, 3, 2)] =>false
So, the process P0 should wait for the resources.

Step 2 :
Checking safety condition for process P1
Need of process P1=need1 = (1, 2, 2)
if needi≤ Available =>[(1, 2, 2) ≤ (3, 3, 2)]=> true
So, process P1 will execute and it is added to safe sequence list {P1}.
Available = Available + Allocation
= (3, 3, 2) + (2, 0, 0)
= (5, 3, 2)
Step 3: Checking safety condition for process P2
Need of process P2= need2 = (6, 0, 0)
if need2 ≤ Available => [(6, 0, 0) ≤ (5, 3, 2)] =>false
So, P2 will wait and it is not added to safe sequence list.

Step 4 : Checking safety condition for process P3
Need of process P3=need3 = (0,1,1)
If need3 ≤ Available =>[(0,1,1) ≤ (5,3,2)]=> true
So, process P3 will execute and it is added to safe sequence list {P1,P3}.
Available = Available + Allocation
= (5, 3, 2) + (2, 1, 1)
= (7, 4, 3)

Step 5 : Checking safety condition for process P4
Need for process P4=need0 = (4,3,1)

if need4 ≤ Available=>[(4,3,1) ≤ (7, 4, 3)]=>True
So, process P0 will execute and it is added to safe sequence list {P1,P3,P4 }
Available = Available + Allocation
= (7,4,3) + (0, 0,2)

= (7, 4, 5)

Step 6 : Checking safety condition for process P0

Need of process P0= need0 = (7,4,3)
if need0≤ Available =>[(7,4,3) ≤ (7, 4, 5)]=>True
So, P0 will execute and it is added to safe sequence list {P1, P3, P4, P0, P2 }.
Available = Available + Allocation
= (7, 4, 5) + (0, 1, 0)
= (7,5,5)

Step 7 : Checking Safety condition for process P2
Need of process P2= need2 = (6, 0, 0)
if need2 ≤ Available =>[(6, 0, 0) ≤ (7, 5, 5)]=>True
So, P2 will execute and it is added to safe sequence list {P1,P3,P4,P2 }.
Available = Available + Allocation
= (7, 5, 5) + (3, 0, 2)
= (10, 5, 7)
Safety Sequence = <P1, P3, P4, P0, P2>

Thus, the requests of the processes will be fulfilled by the obtained safe sequence <P1, P3, P4, P0, P2>.

The Bankers Algorithm :

Now, we have tool for determining whether a particular state is safe or not. When a new request is come, algorithm pretends it is granted and checks whether the resulting state is safe or not. If the resulting state is safe then the request is granted otherwise t is denied as specified in the following steps.

Let Request[i] be the request vector for process Pi.

1. If Request[i] < Need[i], go to step 2. Otherwise, raise an error condition, since the process has exceeded its maximum claim.
2. If Request[i] < Available, go to step 3. Otherwise, process pi must wait, since the resources are not available.
3. Check if the request can be granted safely, by pretending it can allocate the requested resources and then check whether state will be safe or not. If it finds the safe state then the request is granted and if not, then the process must wait till its requested resources can be granted to maintain the safe state. The procedure for

granting the request or pretending to be granted (for testing purpose) is as follows:

Available = Available - Request[i]

Allocation = Allocation + Request[i]

Need[i] = Need[i] - Request[i]

Illustrative Example 5.2 : In the above question 5.1 what will happen if process P1 requests one additional instance of resource type A and two instances of resource type C? Explain whether the request can be safely granted or not.

Solution :

In order to decide whether the request of process P1 can be granted or not, we will use Banker's Algorithm. Initially we assume the request can be granted and try to find a safe sequence. If the safe sequence is found, then resulting system is in safe state and the request of process P1 can be granted.

Request of process P1=request1=1, 0, 2

After granting the requests, we make following changes in the system as discussed above.

Available= Available-request1

Current allocation of process P1= Allocation1=Allocation+request1

Current need of process P1= need1=need1-request1

Process	Allocation	Need	Available
	ABC	ABC	
P0	010	743	230
P1	302	020	
P2	302	600	
P3	211	011	
P4	002	431	

Now, we will use safety algorithm to check whether the current system is in safe state or not.

Step 1 : Check the Safety condition for process P0
Need of process P0= need0 = (7, 4, 3)

If need0 ≤ Available => [(7, 4, 3) ≤ (2, 3, 0)] =>false
so, the process P0 should wait for the resources.

Step 2 :
Check the safety condition for process P1
Need of process P1=need1 = (1, 2, 2)
if need1 ≤ Available =>[(0, 2, 0) ≤(2, 3,0)]=> true
So, process P1 will execute and it is added to safe sequence list {P1}.
Available = Available + Allocation
= (2, 3, 0) + (3, 0, 2)
= (5, 3, 2)

Step 3 : Check the safety condition for process P2
Need of process P2= need2 = (6, 0, 0)
if need2 ≤ Available => [(6, 0, 0) ≤(5,3,2)] =>false
So, P2 will wait and it will not be added to safe sequence list.

Step 4 : Check the safety condition for process P3
Need of process P3=need3 = (0,1,1)
If need3 ≤ Available =>[(0,1,1) ≤ (5,3,2)]=> true
So, process P3 will execute and it is added to safe sequence list {P1,P3}.
Available = Available + Allocation
= (5, 3, 2) + (2, 1, 1)
= (7, 4, 3)

Step 5 : Check the safety condition for process P4
Need for process P4=need0 = (4, 3,1)
if need4 ≤ Available=>[(4, 3,1) ≤ (7, 4, 3)]=>True
So, process P0 will execute and it is added to safe sequence list {P1,P3,P4
}
Available = Available + Allocation
= (7,4,3) + (0, 0,2)

= (7, 4, 5)

Step 6 : Check the safety condition for process P0

Need of process P0= need0 = (7,4,3)
if need0≤ Available =>[(7,4,3) ≤ (7, 4, 5)]=>True
So, P0 will execute and it is added to safe sequence list {P1, P3, P4, P0, P2 }.
Available = Available + Allocation
= (7, 4, 5) + (0, 1, 0)
= (7,5,5)

Step 7 : Check the Safety condition for process P2
Need of process P2= need2 = (6, 0, 0)
if need2 ≤ Available =>[(6, 0, 0) ≤ (7, 5, 5)]=>True
So, P2 will execute and it is added to safe sequence list {P1,P3,P4,P2 }.
Available = Available + Allocation
= (7, 5, 5) + (3, 0, 2)
= (10, 5, 7)
Safety Sequence = <P1, P3, P4, P0, P2>

So the new request is safe and we can immediately grant resource requests of p1.

Check your progress

1. What is a safe state? When does a state is said to be in safe state?
2. What do you mean by a safe sequence?
3. How does the deadlock prevention algorithm works?
4. Explain various methods of handling deadlock.

5.8 DEADLOCK DETECTION AND RECOVER

The method of handling deadlock is employed when, both deadlock prevention and deadlock avoidance algorithm are failed to prevent / avoid a deadlock. This approach detects a deadlock when it is happened and recovers from the deadlock. The disadvantage of this approach is the performance loss due to constant checking for deadlock condition. Also, there is a potential loss of work when a process is aborted or preempted in order to recover the system from the deadlock.

Single Instance of Each Resource Type: If each resource category has single instance, we can use a variation of resource allocation graph called as wait for graph. A wait for graph can be constructed by removing the resources and collapsing the associated edges in the resource allocation graph as shown in Figure 5.6. An arc from process Pi to Pj indicates the process Pi is waiting for a resource which is held by process pj. A cycle in wait for graph indicates the deadlock situation in the system. This algorithm must maintain wait for graph and periodically detects cycle in the graph.

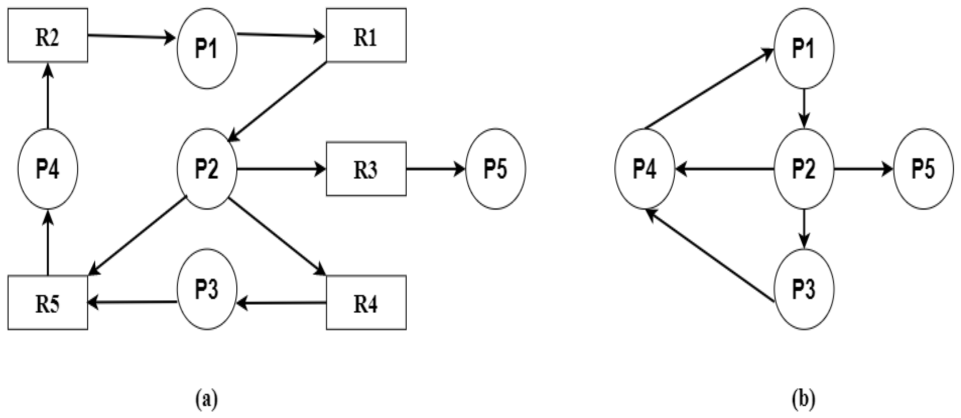


Figure 5.6 - (a) Resource allocation graph. (b) Corresponding wait-for graph [1].

Several Instances of a Resource Type :

When a system has several instances of each resource type, deadlock is detected by the banker's algorithm. In step 1, the Banker's Algorithm sets Finish[i] to false for all i. The algorithm presented here sets Finish[i] to false only if Allocation[i] is not zero. If the currently allocated resources for this process are zero, the algorithm sets Finish[i] to true. This is essentially assuming that if all of the other processes can finish, then this process can finish also. Furthermore, this algorithm is specifically looking for which processes are involved in a deadlock situation, and a process that does not have any resources allocated cannot be involved in a deadlock, and it can be removed from any further consideration.

- Steps 2 and 3 are unchanged
- In step 4, the basic Banker's Algorithm says that if Finish[i] == true for all i, then there is no deadlock. This algorithm is more specific, by saying that if Finish[i] == false for any process Pi, then that process is specifically involved in the deadlock which has been detected.

Recovery from Deadlock :

When the detection algorithm detects presence of a deadlock in a system, there are two options to break the deadlock:

1. **Process Preemption :** In this method, we can terminate processes involved in the deadlock. We can either terminate all the processes that involved in deadlock or terminate processes one by one till the deadlock is removed. Terminating all processes result in loss of partial computations. While, terminating process one by one does not degrades system performance. But, process termination one by one till the deadlock is removed depends on many factors such as priority of process, how long process is running, how many and what type of resources the process is holding.
2. **Resource Preemption :** In this approach, resources are preempted from the processes those are involved in a deadlock. The preempted resources are then allocated to other waiting processes so that, the system cannot be recovered from the deadlock.

5.9 SUMMARY

In summary :

- We discussed the Deadlock problem in detail.
- You learned the four necessary conditions for occurrence of a deadlock.
- You are explained about the Resource Allocation Graph which is very useful for understanding and modelling an algorithm to prevent the deadlock.
- You have understood deadlock prevention algorithm which prevents the system to enter in deadlock condition.

- We discussed deadlock avoidance algorithm in detail.
- At the end, we discussed deadlock detection and Recovery from the deadlock when deadlock prevention or deadlock Avoidance algorithm could not be employed in a system.

5.10 TERMINAL QUESTIONS

1. What are various methods to recover a system from deadlock?
2. What is the usage of Banker's algorithm?
3. How does the deadlock is avoided in a system with single instance of each resource types and multiple instances of each resource types.
4. How does deadlock is avoided through Resource Allocation Graph Algorithm?
5. An operating system uses the Banker's algorithm for deadlock avoidance when managing the allocation of three resource types X, Y, and Z to three processes P0, P1, and P2. The table given below presents the current system state. Here, the Allocation matrix shows the current number of resources of each type allocated to each process and the Max matrix shows the maximum number of resources of each type required by each process during its execution.

	Allocation			Max		
	X	Y	Z	X	Y	Z
P0	0	0	1	8	4	3
P1	3	2	0	6	2	0
P2	2	1	1	3	3	3

There are 3 units of type X, 2 units of type Y and 2 units of type Z still available. The system is currently in a safe state. Consider the following independent requests for additional resources in the current state:

REQ1 : P0 requests 0 units of X,

0 units of Y and 2 units of Z

REQ2 : P1 requests 2 units of X,

0 units of Y and 0 units of Z

Whether the REQ1 can be permitted or Only REQ2 can be permitted or Both REQ1 and REQ2 can be permitted?

6. A computer has six tape drives, with n processes competing for them. Each process may need two drives. What is the maximum value of n for the system to be deadlock free?

7. Can a resource allocation graph have cycles without a deadlock existing? If so state why and draw a sample graph; if no state why not?

8. One method of recovering from deadlock is to kill the processes with the lowest costs of deletion. These processes could then be restarted and once again allowed to compete for resources. What potential problem might develop in a system using such an algorithm? How would you solve this problem?

9. Demonstrate the truth and falsity of each of the following statements.

   a) The four conditions for a deadlock to exist are also sufficient if there is only one resource of each resources type involved in the circular wait

   b) The four conditions for a deadlock to exist are also sufficient if there are multiple resources of each resource type involved in the circular wait.

10. In a system in which it is possible for a deadlock to occur, under what circumstances would you use a deadlock detection algorithm?

11. Which of the following is NOT true of deadlock prevention and deadlock avoidance schemes?

   A. In deadlock prevention, the request for resources is always granted if the resulting state is safe

   B. In deadlock avoidance, the request for resources is always granted if the result state is safe

   C. Deadlock avoidance is less restrictive than deadlock prevention

   D. Deadlock avoidance requires knowledge of resource requirements a priori.

UNIT-VI MEMORY MANAGEMENT

Structure

6.1 INTRODUCTION

The main motive of a computer system is to execute programs or processes. These programs must be brought into the physical memory for their execution. In order to maximize CPU utilization and their response time, OS must bring several processes in physical memory. The effectiveness of a memory management depends on many factors including hardware design of the computer system. In this chapter, we will discuss various memory management schemes from bare hardware approach to paging and segmentation.

6.2 OBJECTIVES

In this unit you will learn following concepts.

- Binding of Instructions and Data to Memory.
- To discuss various memory-management techniques, including paging and segmentation.
- Understands the benefits of a virtual memory system.
- The concepts of demand paging, page-replacement algorithms, and allocation of page frames
- Thrashing - Working Set Window, Page-Fault Frequency

6.3 BACKGROUND

The physical memory which is also known as main memory or RAM is a large array of blocks and each of these blocks is accessed with an address. When going to low level details, the CPU fetches instructions from the main memory based on the value of the program counter. After fetching an instruction from the main memory, the instruction is decoded and operands are fetched from memory for its execution. After execution of the instruction, the results are stored in the memory.

Basic Hardware :

CPU has direct access to only registers and main memory. Any data stored in secondary memory must be first transferred to main memory before its execution. Generally, CPU executes more than one instruction on one clock tick. CPU can access registers very fast usually one clock tick. While, CPU accesses the main memory comparatively slow and takes a number of clock ticks. This can make CPU long waiting if there would be no intermediately memory. Fortunately modern computers are built with very fast cache memory. This cache memory act as an intermediary memory between CPU and main memory. During process execution, the process is first transferred from main memory to cache and then CPU accesses the process directly from cache memory.

Each process has a separate memory space which prevent other process to access its memory space. This is the basis to have multiple processes inside the main memory for concurrent execution. To separate memory spaces of processes, we need to determine legal range of addresses a process can access while other processes can be restricted to access these memory addresses. This scheme can be implemented by using two registers: a base and a limit register. The base register holds starting address of a currently running process while, the limit register holds the size of the range.

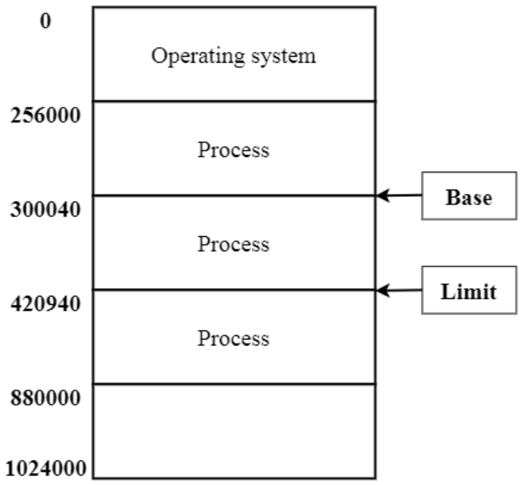


Figure 6.1 : A base and a limit register define a logical address space

For example in Figure 6.1, of the base register holds an address 300040 and the limit register hold a range of 120900, the associated process can access all addresses between 300040 to 420940. During the course of execution of a process, every address generated in user mode is compared with base and limit register addresses by CPU hardware support as shown in Figure 6.2.

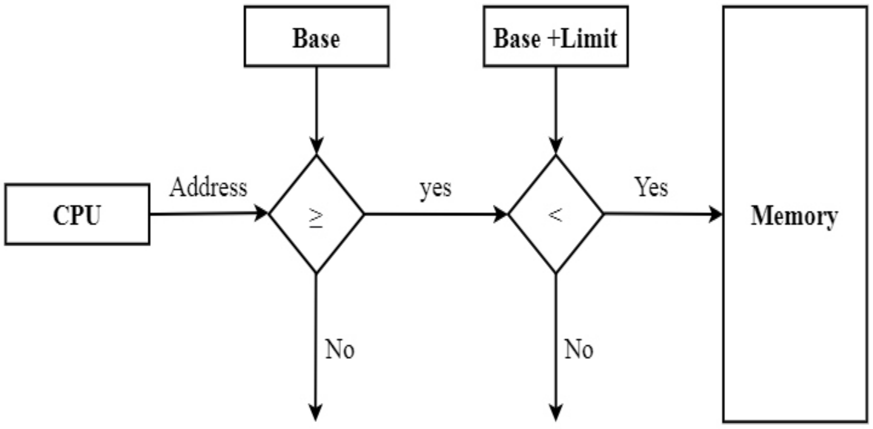


Figure 6.2: Hardware address protection with base and limit registers [1].

An invalid address or any attempt by a process executing in user mode to access OS memory address or they process address would generate trap to the operating system as shown in Figure 6.2. The base and limit register can only be loaded and changed by the OS via privileged instruction that run only in kernel mode.

6.4 ADDRESS BINDING

Logical and physical address space: During the execution of a process, CPU generates logical address. This logical address is mapped to a physical address which is the actual address in the main memory the process would be loaded. This translation of a logical address to a physical address is done by memory management unit of OS. The logical address is also known as a virtual address. A set of all logical addresses used by a process is known as logical address space of the process. And, a set of all corresponding physical addresses known physical address space of the process. A user process never sees physical addresses. A user process works entirely in logical address space, and any memory references or manipulations are done using purely logical addresses.

Address Binding : is a process of binding logical address space to physical address space. This address binding can be done in three ways as shown in Figure 6.3. The compile time and load time address binding generate identical logical and physical addresses. While the execution time address binding scheme, have different logical and physical addresses.

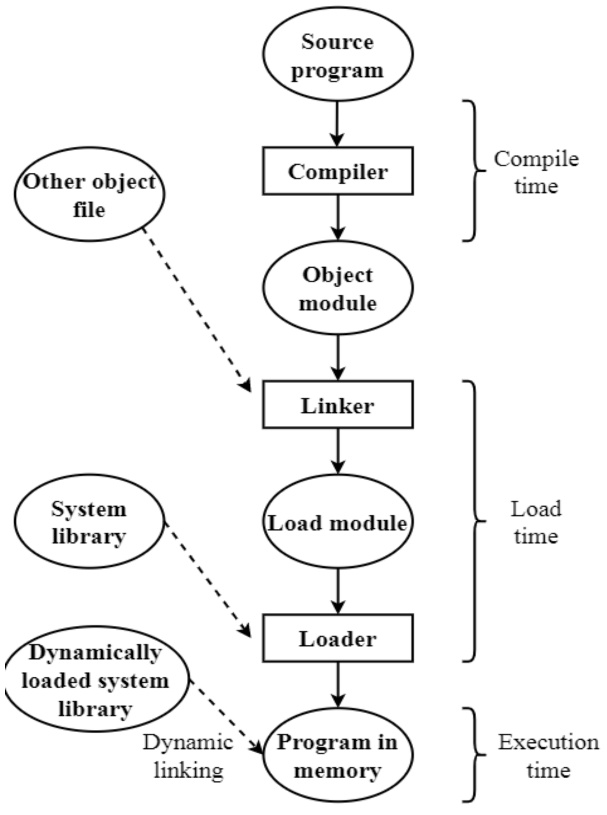


Figure 6.3 : Multistep processing of a user program [1].

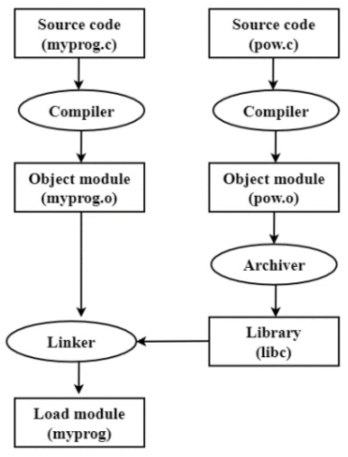
Compile time : If at compile time, compiler knows where a program reside on physical memory then absolute address is generated which is the actual physical memory location. For example, at compile time, compiler knows that the user program will reside at location L, then generated compiler code starts from location L in the physical memory and extends from there. But if the generated absolute address is already occupied by another process then we need to recompile the program.

Load Time : If at compile time you do not know where to load the user's program then compiler generate relocatable addresses relative to start of program (such as "20 bytes from the beginning of this program"). At load time, loader translates relocatable addresses to absolute addresses. Loader adds the base address of the program to all logical addresses of the program to generate relocatable addresses. If the base address of the program changes, we need to reload the program.

Execution Time : If a program moves from one memory location to another during its execution then binding is delayed until execution time. This method is used by most modern operating systems.

Dynamic Loading : Instead of loading an entire program into physical memory, dynamic loader loads each routines or a part of program when it is required. So unused routines are never loaded into the physical memory unless they are required. Dynamic loading saves memory usage and also makes faster program start-up.

Dynamic Linking : In static linking, all the library modules are fully included in executable modules. This wastes both disk space and main memory usage because, executable modules include copy of all the required routines from the library. Figure 6.4, 6.5 briefly describe how load module is created and how does static linking and loading take place. However in dynamic linking, a stub is included in executable module. Stub is a small program that indicate how to locate a library routine or how to load library routine when it is not present in the memory. Next time, when that particular library routine are encountered, there is no need for dynamic linking again since the library routine is already present in the memory. One more advantage is that a library can be replaced with new version and all the program that reference the library can use the new version of the library.



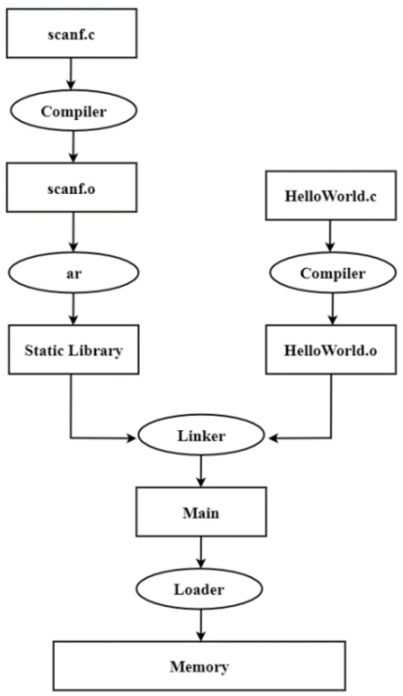


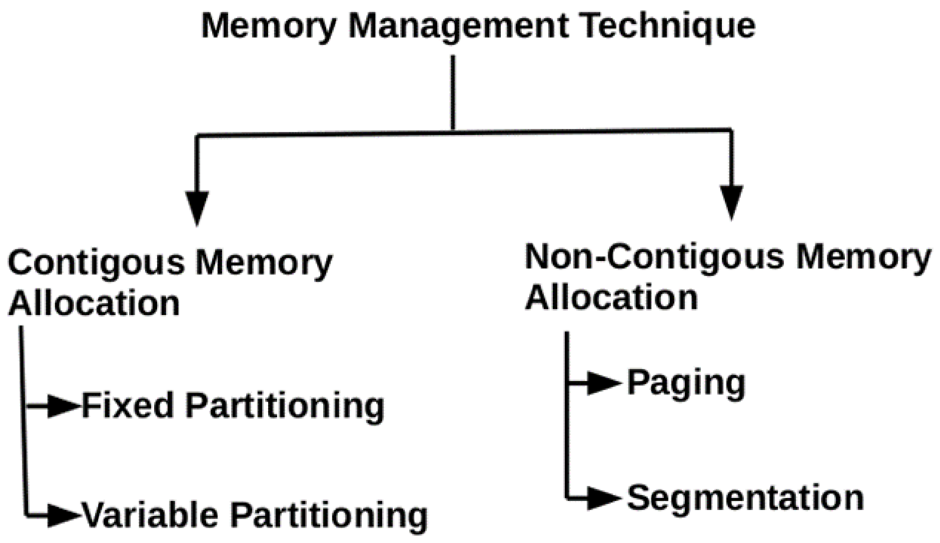
Figure 6.4: Creating a load module [1]. Figure 6.5 Normal linking and loading [1].

Check your progress

1. What do you mean by address binding?
2. Briefly explain the types of address bindings.
3. Discuss the basic hardware supports requires for memory management?
4. What do you mean by dynamic loading and dynamic linking?

6.5 MEMORY MANAGEMENT TECHNIQUES

Broadly, we can divide memory management techniques into two categories: contiguous memory allocation and non-contiguous memory allocation. In contiguous memory allocation, each user process is allocated to single contiguous section of memory. While in non-contiguous memory allocation each process can be allocated to different section of memory. The ultimate goal of the memory management unit is an efficient utilization of memory which lead to minimum internal and external fragmentation which we will discuss in next section.


Memory Management Technique
Contigous Memory Allocation
Non-Contigous Memory Allocation
Fixed Partitioning
Variable Partitioning
Paging
Segmentation

6.6 CONTIGUOUS MEMORY ALLOCATION

This is an early method of memory Management. The main memory holds both operating system and user processes. Usually, memory is divided into two party t. One part of the memory holds operating system, while other part holds users processes. Generally, the OS resides at lower memory section of the main memory and user processes reside at high memory section. In contiguous memory allocation, a process is loaded to single contiguous memory section.

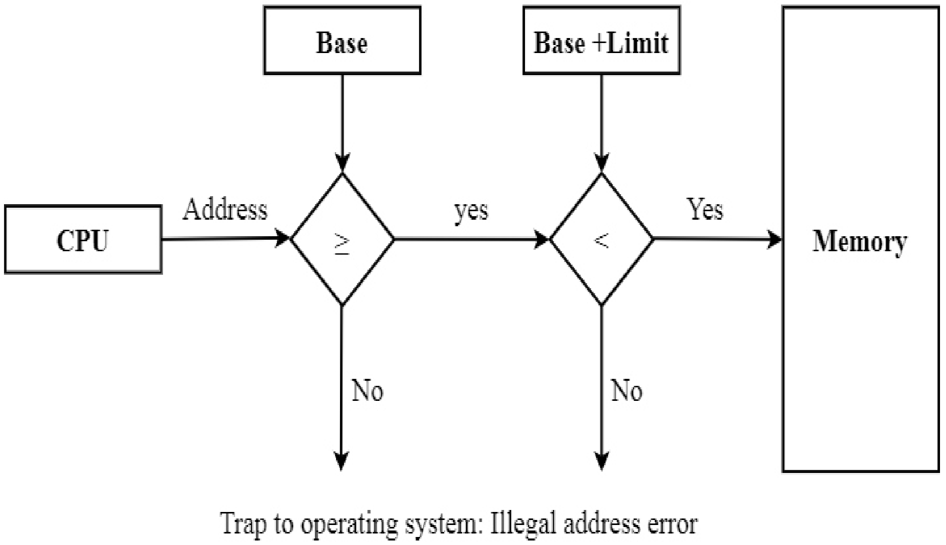


Figure 6.6: Hardware address protection with base and limit registers [1].

Before discussing the allocation of main memory to different processes, we should first discuss how we prevent one process from accessing memory of another processes. This is achieved by using two registers: a relocation register and a limit register (as shown in figure 6.6). The relocation register holds starting address of a process in main memory while, the limit register specify the limit or size of memory from starting address. Each logical address of a process generated by CPU must falls within the specified range given by limit register. When the CPU scheduler selects a process from the ready queue, the dispatcher loads valid values in these registers as a part of context switch. Every address generates by the CPU is checked against the values in the registers. The memory management unit dynamically maps logical addresses to physical addresses by adding the value of relocation register to the logical address.

Memory allocation :

Fixed partitioning scheme: Now, we are ready to discuss memory allocation to processes. One of the simplest memory allocation scheme is fixed sized partition where the memory is divided into several equal sized partitions. Here, the degree of multiprogramming is fixed by the number of partitions. CPU scheduler selects one process from ready queue and places it into one of the available partition. When the process terminates, the partition is freed and it becomes available for other processes. This scheme is used in IBM OS/360 and no more used now.

Variable partitioning scheme: Another contiguous memory allocation scheme is variable partitioning scheme which is more efficient than fixed partitioning scheme. In this scheme, OS maintains a table that keep track of available and occupied sections of memory. Initially, a large block of memory also called as a large hole is available. As the processes execute, and memory is allocated and deallocated, several holes of variable sizes are created. These holes are scattered throughout the main memory. At any time, CPU scheduler selects a process from the ready queue and OS searches for a sufficient large hole. If a hole is available to accommodate this process, then it is loaded into that hole. Otherwise, it waits till a large hole in the memory becomes available to satisfy its memory requirement. Meanwhile, some other process is selected from the ready queue and loaded into a hole which is large enough to accommodate it.

For example, Figure 6.7 shows effect of process allocation in variable partitioning in 64 Mbytes of main memory. Initially, main memory is empty, except for the OS (a). Then three processes 1,2,3 are loaded in, starting where the operating system ends and occupying as much space as required by each process (b, c, d). This results a "hole" at the end of memory which is too small for a fourth process. At some point, the process 2 in memory is not ready. The operating system swaps out process 2 (e), and makes sufficient space to load a new process 4 (f). This creates another small hole as the process 4 is smaller than process 2. Later, a point is reached at which process 1 in main memory is not ready (or in blocked state), but process 2 (wakeup from blocked state) is available. Since there is insufficient space in main memory for process 2, the operating system swaps out process 1(g) and swaps in process 2 (h) back into main memory.

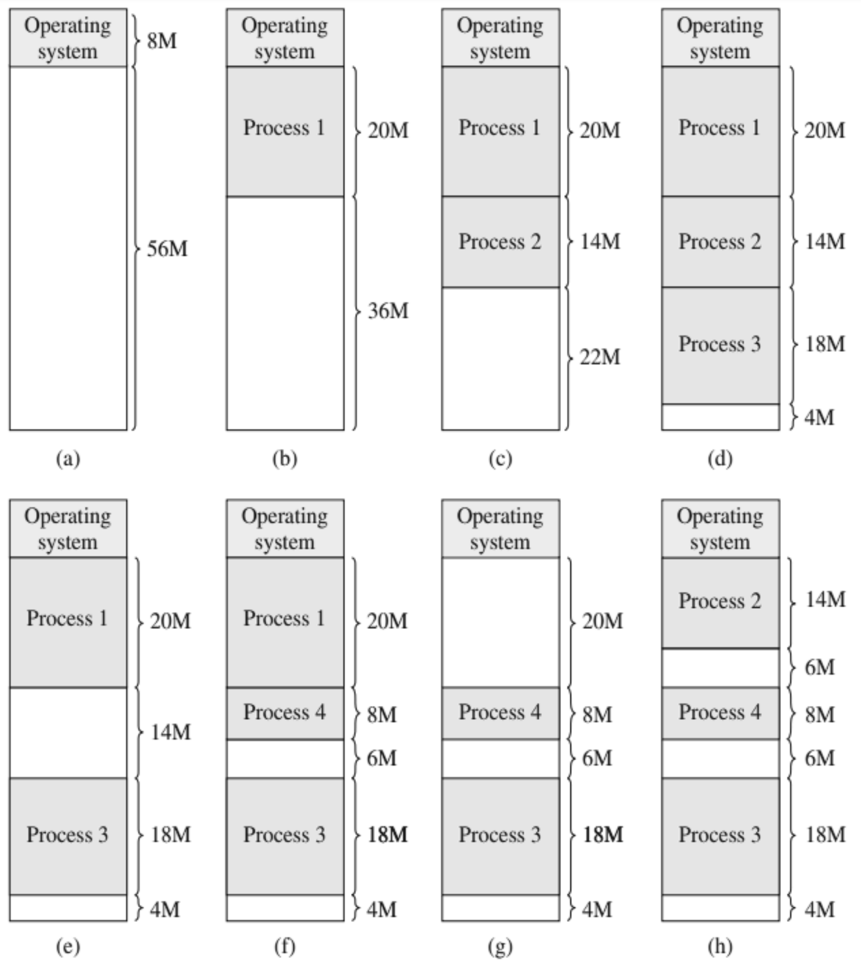


Figure 6.7 : Effect of Variable Partitioning[4]

There exists name strategies to search for a sufficiently large hole for a process. The most commonly used strategies are: first fit, best fit and worse fit.

First fit : It searches for a suitable hole for a process from beginning of a list of holes and stops as soon as it finds the first hole that is large enough to accommodate this process. This way it allocates the first available hole from the set of holes to a process.

Best fit : It searches the entire list of available holes to select a smallest hole that can accommodate a process. It allocates a smallest hole to a process.

Worse fit : It also searches entire list of holes to pick up a largest hole for a process which can accommodate a process. It allocates a largest hole to a process.

The first fit and best fit are faster and storage efficient compared to worse fit. Generally, the first fit does faster allocation of hole to a process than best fit.

Illustrative Example: Given five memory partitions of 100Kb, 500Kb, 200Kb, 300Kb, 600Kb (in order), how would the first-fit, best-fit, and worst-fit algorithms place processes of 212 Kb, 417 Kb, 112 Kb, and 426 Kb (in order)? Which algorithm makes the most efficient use of memory?

First-fit :

212K is put in 500K partition

417K is put in 600K partition

112K is put in 288K partition (new partition 288K = 500K - 212K)

426K must wait

Best-fit :

212K is put in 300K partition

417K is put in 500K partition

112K is put in 200K partition

426K is put in 600K partition

Worst-fit :

212K is put in 600K partition

417K is put in 500K partition

112K is put in 388K partition

426K must wait

In this example, best-fit turns out to be the best.

Fragmentation : Fragmentation is a situation in which memory space is not utilized efficiently which lead to wastage of space and degradation in system performance. This is a weakness of any memory allocation strategies. Fragmentation is of two types: internal and external fragmentation. Both the first fit and best fit strategies suffers from external fragmentation. The first fit can perform better in some systems while best fit could be better for other systems.

External fragmentation : When processes are allocated and deallocated during their executing, a large memory hole is broken down into small many small holes. External fragmentation exist when these holes are not sufficient individually to satisfy memory request of a process. In worse case of the external fragmentation exist when there will be a hole or a waste block of memory between every two processes.

Internal fragmentation : Consider the fixed partitioning scheme where the main memory is divided into several equal sized partition. Suppose, a process require memory less than the size of the partition. When the process is allocated memory into any of available partition, a small amount of memory space will be wasted. For example, if the partition size is 1024 bytes and the process requires 1022 bytes, then a hole of 2 bytes will get wasted. This extra 2 bytes of memory space is internal fragmentation.

Solution of External fragmentation : One solution of external fragmentation is compaction. In compaction, the contents of the memory are shuffled so that all free block of memory or holes comes together at one end to form a big hole. While on other end of memory, all occupied blocks of memory are shifted. Compaction is a costly scheme and it is not always possible. Another solution to external fragmentation is to allow logical address space of a process to be non-contiguous. Thus, a process can be allocated in multiple parts to anywhere in the available memory. This technique is implemented through segmentation and paging which we will discuss next.

Check your progress

1. How does external fragmentation different from internal fragmentation?

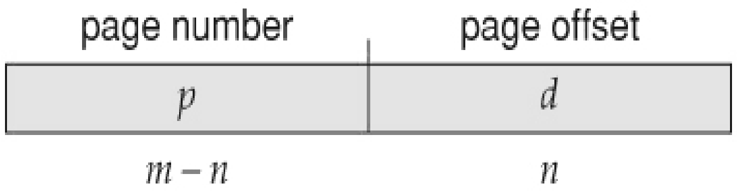
2. What is compaction?
3. Describe the three strategies of memory allocation to a process.
4. How does the contiguous memory management is implemented?
5. Explain the following terms:
    a) First fit
    b) Best fit
    c) Worse fit
6. How does the fixed sized partition differs from variable position scheme?

6.7 PAGING

Paging is another memory management scheme which offers non-contiguous memory allocation. The major problem with the segmentation is external fragmentation. This problem of external fragmentation also there in backing store (swap space inside secondary storage) secondary storage due to the result of swapping. However, the Paging does not suffers from this problem and there is no need for compaction. Due to its advantages over earlier methods, paging is used in modern operating system in various form.

Basic method: In paging, a process is broken into several pages, each of same size. The main memory is also broken into equal sized blocks called frames. Each of these frames is of same size as that of pages. When a process starts its execution, its pages are loaded into any available frames of main memory. With this scheme, a process having logical address space of 2^32 can be executed even though the system has less than 2^32 physical address space or memory (memory is bytes addressable).

When CPU generates a virtual address for a process, it consists of two parts: a page number p, and offset d.



The page number is used to locate associated frame number from its page table as shown in Figure 6.8.

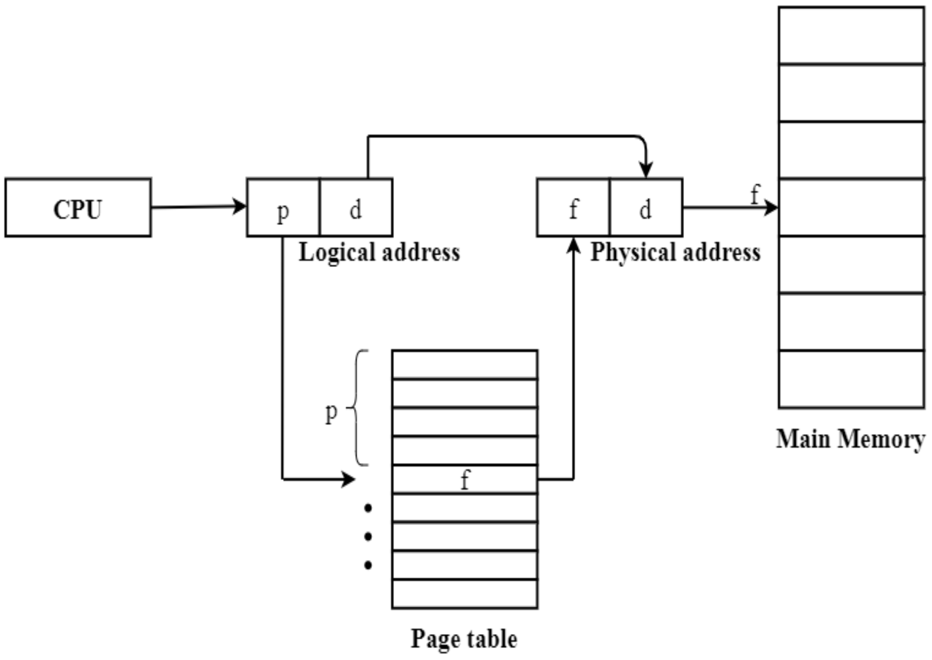


Figure 6.8 : Paging hardware [1].

The page table contains entries for each page of the process along with its associated frame number. The offset specifies the address of a word within the page. Mappings of logical memory pages of a process to frames in physical memory are shown in Figure 6.9.

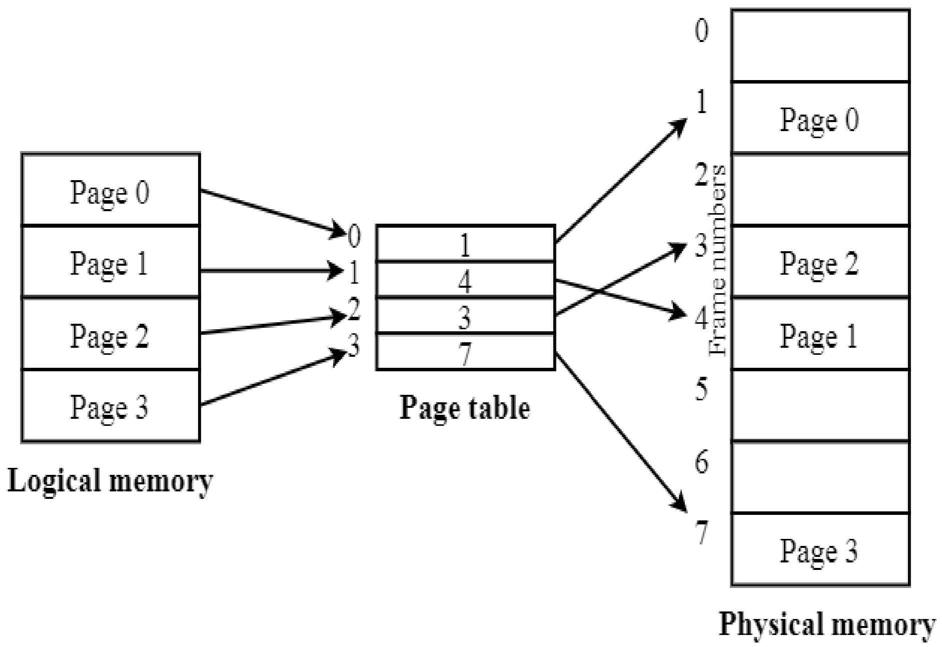


Figure 6.9: Paging model of logical and physical memory [1].

The page size is in the power of 2 which typically range from 512 words (or bytes in byte addressable system) to 1G words. The page size as the power of 2 makes the address translation easier. If a logical address space of a process is 2^m words and page size is 2^n words, m-n bits of the logical address would be used to find the page number in the page table and n bits would be used to locate the word in its frame. Thus a logical address has following format. Here, p is used to index into a page table and offset d is displacement within the page. For example, consider a system with 32 word memory and 4 words page size as shown in Figure 6.10.

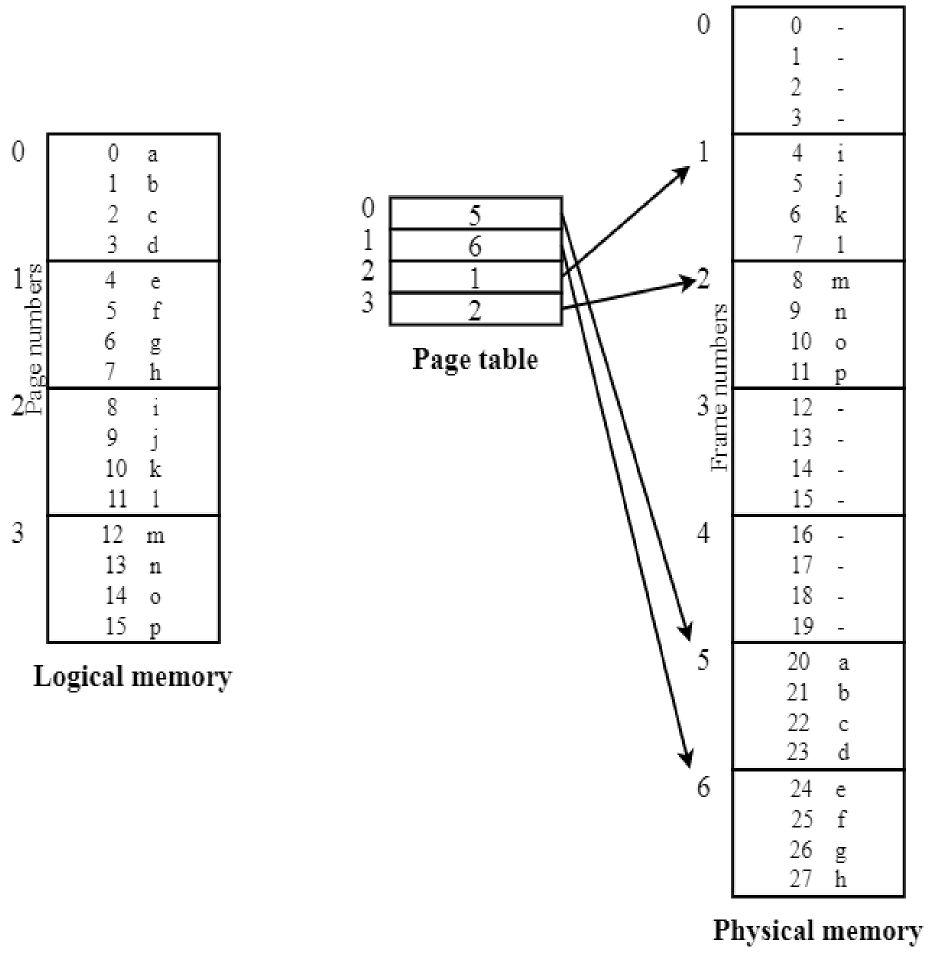


Figure 6.10 Paging example for a 32-byte memory with 4-byte pages [1].

Here, logical addresses are n=2 bits and number of bits to address pages are m=4 bits. The Figure shows mappings of logical addresses containing page numbers and offsets to corresponding memory frames. The logical address 0 is page number 0 and offset 0. By Looking into the page table, we find that page 0 resides at memory frame number 5 and the logical address 0 is mapped to physical address 20 = 5*4+0. Similarly, the logical address 11 is page 2 and offset 3 which is mapped to memory frame 1 at physical address 7 = 1*4+3. Please note that the page and frame numbering starts with 0.

A 32 bit CPU, generally contains the page table entries of 32 bits (for each frame). The 32 bit frame number in the page table can point to 2^32 or 4 G memory frames. If the page and frame size is 4k or 2^12 words, the system can point to 16 TB or 2^44 words. Thus, a 32 bit CPU uses 32 bit address for every process which can point to 2^32 words or 4GB process size. When CPU scheduler selects a process from ready queue, its size is expressed by the number of pages it contains. Each page requires one memory frame. So, if a process consists of n pages, n memory frames should be available to execute it. The first page is loaded into one of the available memory frames and the frame number is put into page table. Similarity the second page is also loaded into one of the available memory frame and this mapping is put into page table as an entry. This will continue for all n pages as shown in Figure 6.11.

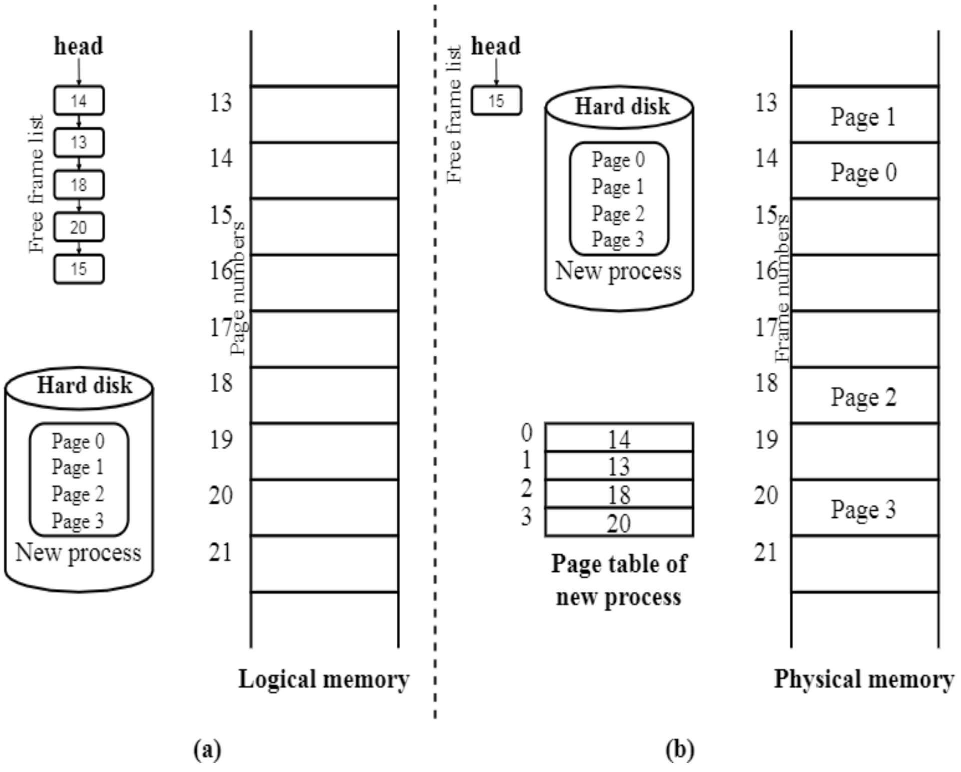


Figure 6.11 Free frames (a) before allocation and (b) after allocation [1].

OS maintains a copy of page table in memory for a process it want to execute. This copy of page table is used for translation of logical addresses to physical addresses during its execution. CPU dispatcher also uses this copy to define hardware page table when it allocates process to CPU. OS should be aware of which memory frames are allocated, which are available, number of memory frames etc. For this, it maintains a frame table which contains details for every memory frame. The frame table indicates whether a frame is occupied or available, which page and which process has occupied the frame.

The paging scheme suffers does not suffer from external fragmentation. However, there is an internal fragmentation due to the last page of a process, if memory requirement of the last page does not exactly equal to frame size. For example, if a process size is 16400 words and page size is 1024 words, the process will require 16 pages + 16 words. Thus, the process will requires total-17 pages, however the last 17th page will only contain 16 words data and remaining will be unoccupied. When this last page will be allocated a frame, a space of 1024-16=1008 words in the frame will be unoccupied which will cause internal fragmentation. In worse case a process could contain n pages + 1 word which would require total n+1 frames. This suggests that smaller page size is desirable. But, this would result in too many pages which would require too much overhead to maintain page table entries. This overhead reduces as we increase page size. Also, disk I/O will be more efficient when the data size to be transferred is larger. Page size has grown over the time as size of processes, main memory, and data sets increases. In modern operating systems, usually page sizes are 4 k words and 8 k words while, some operating systems have even larger than that.

Hardware Support :

OS maintains separate page table for each process. A page table is brought into the memory during execution of a process and a page table base register (PTBR) points to this page table. The value of the PTBR is stored in PCB on the process and its value changes by CPU dispatcher. The major problem with this scheme is that, it requires two memory access to access a desired memory location (or translation of a logical address to physical address). Thus, the memory access is slowed by the factor of 2 for access to each physical address. One memory access requires to access the page table in memory based on PTBR value. The page table gives corresponding frame number. The frame number is used to access desired memory location which requires this another memory access. This problem can be addressed by using a set of dedicated registers to maintain a page table. These registers are built with high speed logic. Each mapping of logical to physical address must go through these registers. The values of these registers are loaded by CPU dispatcher during execution of a process. However, this method works satisfactory only if the page table contains small entries such as 256 entries. But the size of page table growing continuously and the modern operating systems contain around millions of entries in a page table. So it is not feasible to maintain the page table using these fast registers.

Avoiding extra memory access: A standard solution to above problem is to use a small high speed associative memory called TLB (translation look-aside buffer) as shown in figure 6.12. It consists of two parts: a key and a value. All comparison are done on keys simultaneously. The TLB contains only few page table entries. When a logical address of a process is presented to TLB, the page number is compared with entries in the TLB. If the match is found, its frame number is accessed and used to get its physical memory location. If the page number is not found in the TLB, OS looks into the page table in the memory. The frame number associated with the page is then accessed to get its actual physical memory location. We then add this frame entry in the TLB, so that next time when the page is again referenced, the corresponding frame number is quickly accessed.

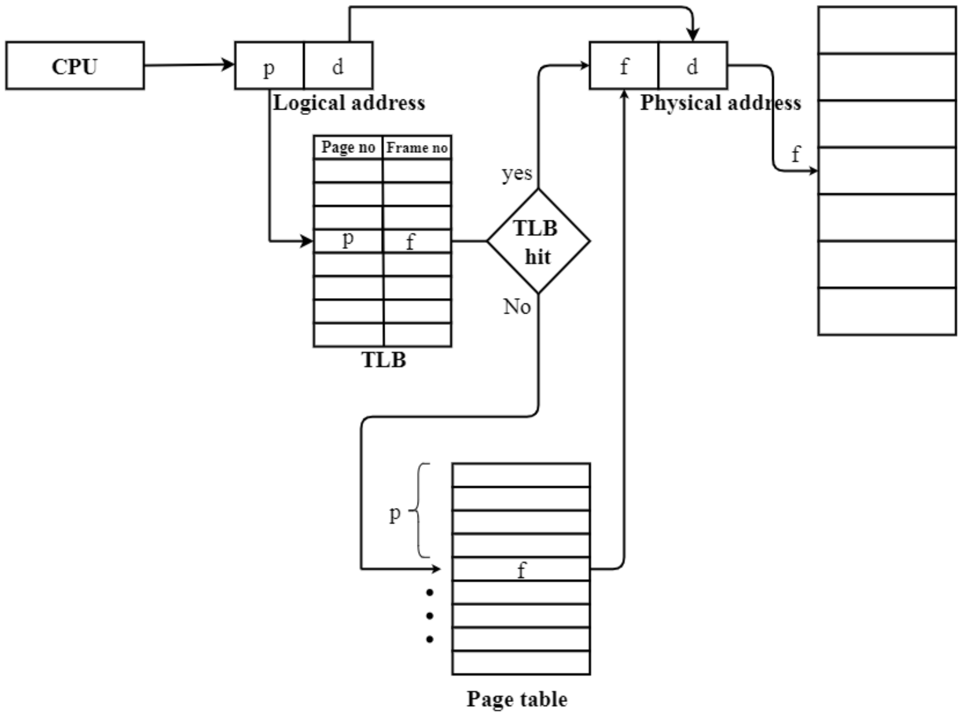


Figure 6.12 : Paging hardware with TLB [1].

If the TLB is full of entries, then one of its entries must be removed to add above entry in TLB. Existing entries are replaced with new entries using a Replacement strategy that may vary from LRU (least recently used) to random. Every time when a new process is executed, the TLB must be cleared to accommodate new entries of the page table. This ensures that the new executing process could not access old entries in the TLB that could lead to wrong address translation. The flushing of TLB every time when a new process is executed can be avoided by storing an address space identify (ASIDs) along with key and value in each entry of TLB. The ASID uniquely identifies each process in a system. So, if a new process executes, its ASID is matched with the one stored in the TLB. If it matches, the page number is then matched and its frame number is retrieved.

Illustrative Example: Consider a paging hardware with a TLB. Assume that the entire page table and all the pages are in the physical memory. It takes 10 milliseconds to search the TLB and 80 milliseconds to access the physical memory. If the TLB hit ratio is 0.6, what is the effective memory access time (in milliseconds) if there is no page-fault?

Solution: As both page table and page are in physical memory

Effective memory access time

= hit ratio * (TLB access time + Main memory access time) + (1 - hit ratio) * (TLB access time + 2 * main memory time)

= 0.6*(10+80) + (1-0.6)*(10+2*80)

= 0.6 * (90) + 0.4 * (170) = 122 ms.

Protection: In paging environment, a memory protection is achieved by using a valid-invalid bit in each entry in a page table. If this bit in the page table entry is set to valid, the page is in the logical address space of the process. And if, it is set to invalid, the associated page is not in the logical address space of the process. For example, consider a system with 14 bit address space and a page size of 1024 words as shown in Figure 6.13.

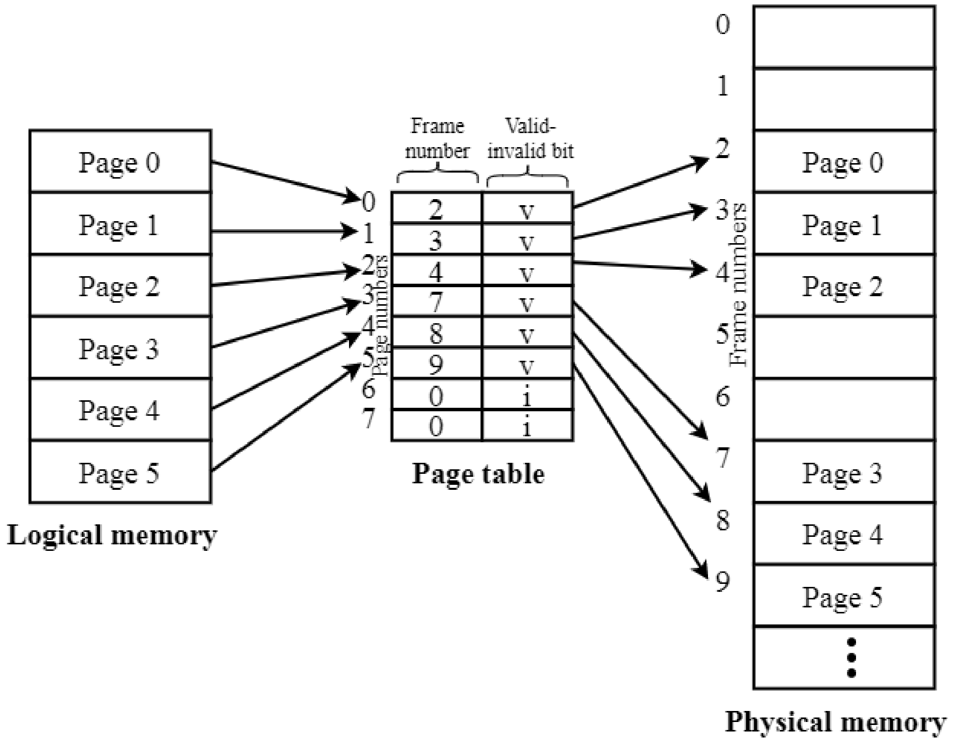


Figure 6.13: Valid (v) or invalid (i) bit in page table [1].

Suppose a process is of size 10468 words. The process will require total 6 pages =ceiling (10468/1024). While the number of entries or pages in page table is 8 = 2^14/1024. The pages 0,1,2,3,4,5 contain valid frame number since these pages belong to the process. So the corresponding valid-invalid bits are set to valid. While, the pages 6,7 does not belong to the process, so their valid-invalid bit entries are set to invalid.

Rarely, a process uses all the pages in a page table. Many of entries in the page table are invalid. But they still occupies valuable space in memory. This can be avoided by using page table length register (PTLR) that points to the length of the page table. Every logical address of a process is checked against this register value to verify the address is in valid range of the process

Shared pages : Another advantage of paging is sharing of a common code among more than two processes. The common read only code that never changes during execution can be shared by multiple processes. The pages containing common code that belong to several processes have same entries in their page table and thus they are all mapped to same memory frames. For example, consider a system with 40 user are using text editor as shown in Figure 6.14.

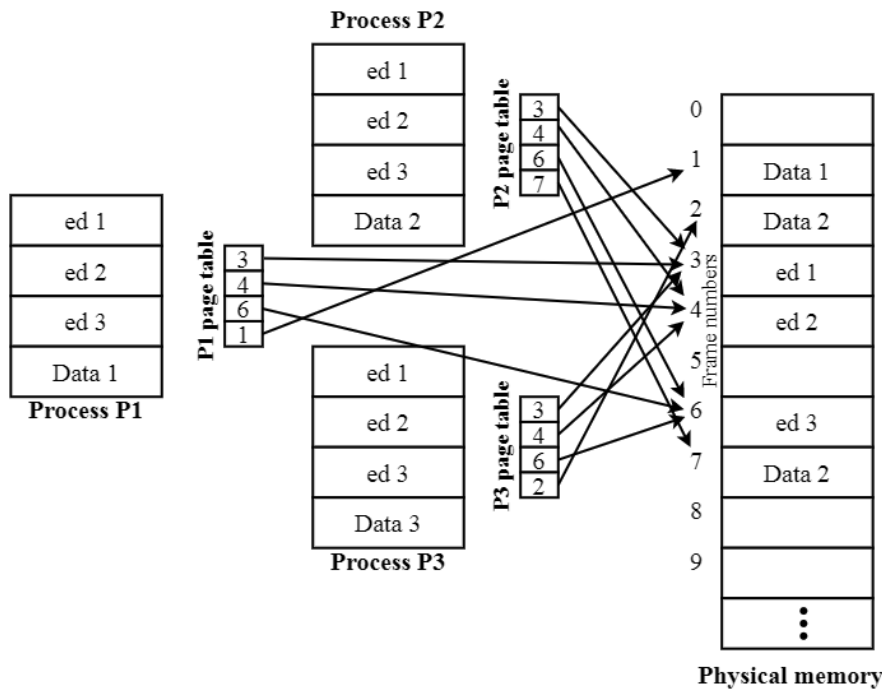
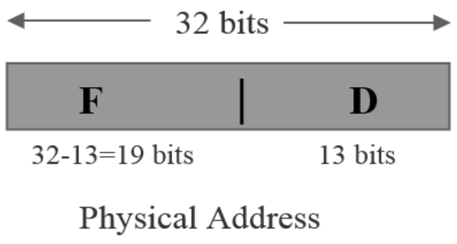


Figure 6.14: Sharing of code in a paging environment [1].

The text editor requires 150 KB for code and 50 KB for data. Thus, to support 40 user, the system will requires total 8000 KB memory space. But, with page sharing scheme, the system will require only 2150 KB memory space. Assume that the page size is also 50 KB. Now, each user process contains 4 pages (200/50) of which 3 are for common code and 1 page for their data. So, the pages containing common code needs only three memory frames while the page containing data specific to each process require one frame each.

Illustrative Example : computer system implements 8 kilobyte pages and a 32-bit physical address space. Each page table entry contains a valid bit, a dirty bit, three permission bits, and the translation. If the maximum size of the page table of a process is 24 megabytes, what is the length of the virtual address supported by the system?

Solution :



210 = 1K

220 = 1M

230 = 1G

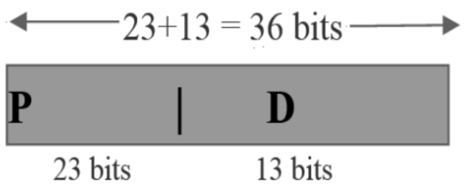
240 = 1T

Page Size= 8KB=23 * 210 =213 words or byte

Number of bits for offset= 13 bits
For 32 bit physical address, 32 - 13 = 19 page frame bits must be there in each PTE (Page Table Entry).
We also have 1 valid bit, 1 dirty bit and 3 permission bits.
So, Page Table Entry Size = 19 + 5 = 24 bits = 3 bytes.

Given in question, maximum page table size = 24 MB
Page table entry size = Total number of pages in page table * size of an entry
Total number of pages in page table= No. of PTEs = 24 MB / 3 B = 223 = 8 M

Number of bits required to address pages in page table (P) = 23 bits



Virtual Address

So, length of virtual address =23+13= 36 bits (assuming byte addressing)

Check your progress

1. Consider a paging hardware with a TLB. Assume that the entire page table and all the pages are in the physical memory. It takes 10 milliseconds to search the TLB and 80 milliseconds to access the physical memory. If thse TLB hit ratio is 0.6, what is the effective memory access time (in milliseconds).
2. Consider a system with byte-addressable memory, 32 bit logical addresses, 4 kilobyte page size and page table entries of 4 bytes each. What is the size of the page table in the system in megabytes?
3. Does paging scheme suffers from external fragmentation?
4. Paging suffers from ___________ _______________ and Segmentation
   suffers from __________ ____________.

6.8 STRUCTURE OF THE PAGE TABLE

The modern computer systems support a large logical address spaces which typically vary from 2^32 to 2^64 words. This results increase in size of the page table and it will take more main memory space. For example, consider a system with 32 bit logical address (2^32 logical address space) and it supports the page size of 4k or 2^12 words. This system will require total 2^20 =2^32/2^12 entries (around 1 million entries) in the page table. If a page entry is 4 byte, then the page table will take 4 M words = 2^20*4 bytes or 4 MB (if the memory is byte addressable) space. But, we do not want to allocate this much block of memory contiguously. One solution to this problem is to divide the page table into several page tables which we will see next.

Hierarchical paging : A two level paging is most basic structure in hierarchical paging scheme. In this structure, a page table is divided into two page tables: outer and inner page table as shown in Figure 6.15.

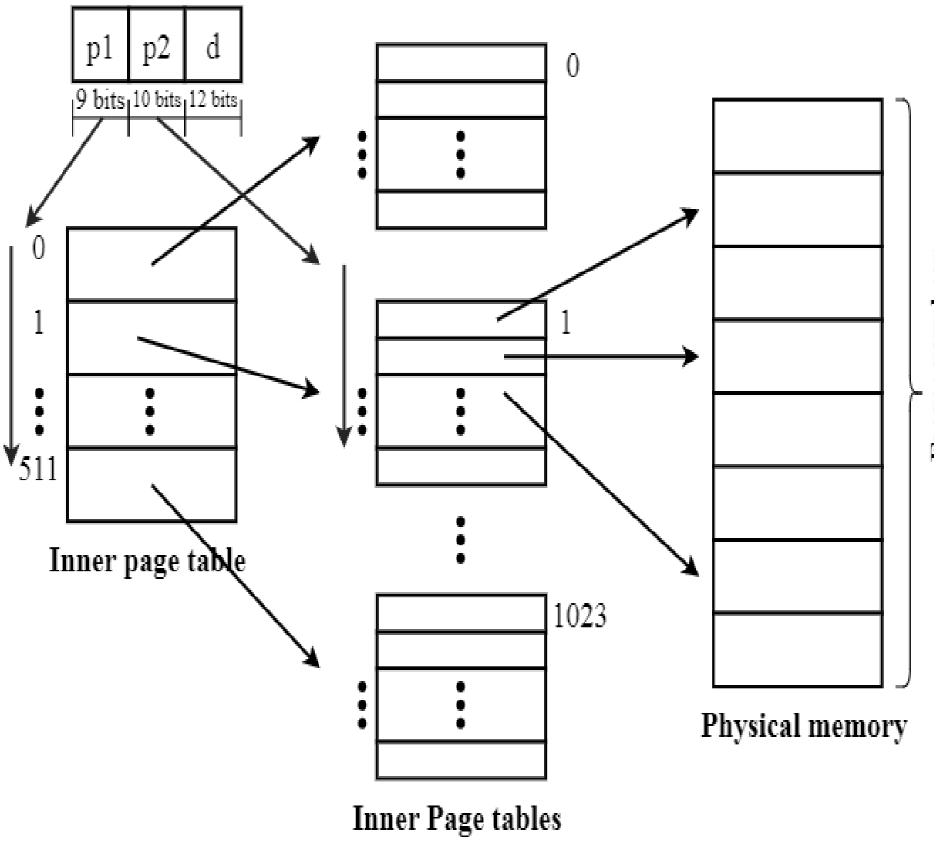
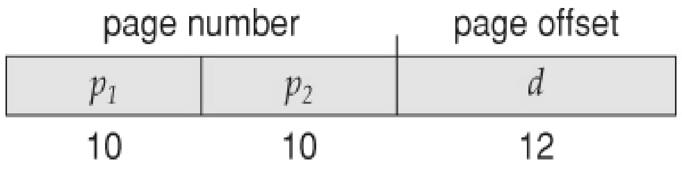


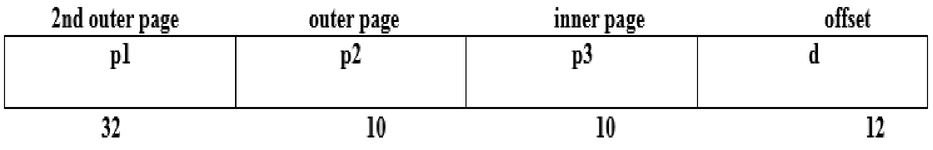
Figure 6.15 : A two-level page-table scheme [1].

In this scheme, a logical address generated by CPU is divided into three parts. One part of the logical address is used to look into the outer page table is used to each entry in outer page table is used to access entries in inner page table. For example, consider again the system that supports 32 bit logical address and the page size of 4 K words. Under two level paging scheme, a logical address generated by CPU is divided into three parts as shown below.



The first part p1 is 10 bit page number which is used to index into pages of outer page table. The second part p2 is also 10 bit page number which gives displacement within a page in inner page table. The third part d is 12 bit page offset which is added to the address of the frame to get the desired memory location.

For a 64 bit system, the two level paging scheme will not be suitable. To illustrate this, consider a 64 bit system with page size of 4 K words. Under 2 level paging, the outer page table will contain 2^42 entries and the inner page table will contain 2^10 entries. If each entry in outer page table is 4 bytes, then the outer page table will require 14 GB= 2^42*4. This is significant large page table size. To overcome this problem, we can further divide the outer page table giving the level paging as shown below.



Under three level paging scheme, the outer page will contain 2^32 entries which will result in 16 GB= 2^32*4 bytes size.

Disadvantage : The major problem with hierarchical paging scheme is that the number of memory access increases with increase in number of page tables. So, the hierarchical paging scheme is not suitable for such architectures.

Illustrative Example : Consider a system using 2 level paging applicable page table is divided into 2K pages each of size 4KB. If size of physical memory is 64 MB which is divided into 16K frames. Calculate the length of logical and physical address assuming memory is byte addressable.

Solution :

Number of pages or entries in outer page table size= 2K=2*210= 211.

So the number of bits required to address each pages of outer page table = 11 bits

Inner page table size=4KB=22*210=212 words. (Since memory is byte addressable, 1 byte is equivalent to 1 word)

So the number of bits required to address each pages of inner page table = 12 bits

Size of physical memory = 64 MB= 26*220 = 226 words. (Since memory is byte addressable, 1 byte is equivalent to 1 word)

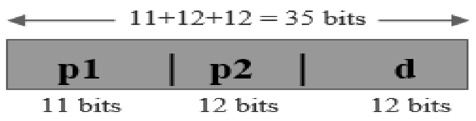
Total number of physical memory frames= 16K=24*210= 214

So, the number of bits required to address each frame in physical memory= 14 bits

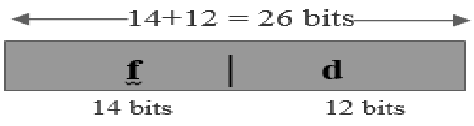
Physical memory frame size = Size of physical memory/total number of physical memory frame

= 226/214 = 212 words

So, number of bits required for page offset = number of bits required for frame offset= 12 bits



logical address



physical address

Inverted page table : Generally, each process has its own page table for logical to physical address mapping. The modern computer system usually have around millions of entries in a page table. We have seen earlier that as number of entries in a page table increases, the size of page table also increases. The large page table consumes more space in main memory. This problem can be solved by using inverted paging as shown in figure 6.16.

In inverted paging, the OS maintains a single page table for all process. So, all process are using a same page table. The number of entries in the inverted page table are equal to the number of frames in main memory. Since, each process uses the same page table, an address space identifier is required to uniquely identify each process's page. Each entry in the inverted page table also contains a process ID as an address space identifier.

<process id, page number, offset>

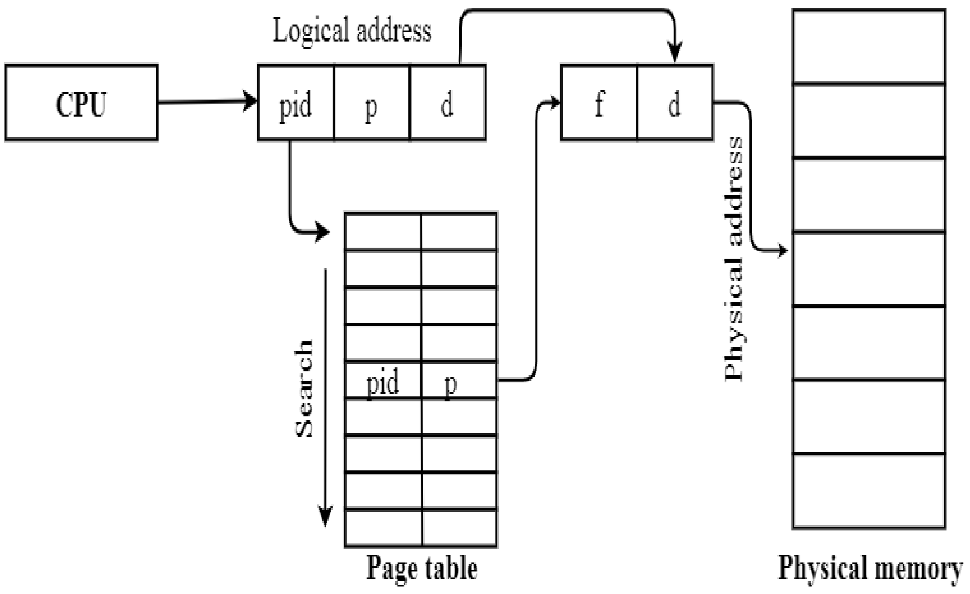


Figure 6.16 : Inverted page table [1].

When CPU generates a logical address, the part of the logical address containing process ID and page number are used to search against process IDs and page numbers in inverted page table. If a match found, associated frame number is retrieved. The frame number along with the offset are then mapped to actual physical memory location.

Disadvantage : The major disadvantage with inverted are long search time into the page table and implementation of shared memory. Since, in the inverted page table there is a single entry for each memory frame, but for implementing shared memory multiple entries of a memory frame for other process must be present. Inverted page table and its variations are used in 64 bit PowerPC, UltraSPARC and the IBM RT.

Illustrative Example : In a 64 bit machine, with 2GB RAM and 8KB page size, how many entries will be there in the page table if it is inverted page table?

The number of entries in page table of an inverted page table is equal to number of frames in the main memory.Number of frames in RAM= Size of RAM/page size = 2 ∗ 230/(23 ∗ 210) = 218=256 K

6.9 SEGMENTATION

Segmentation is a memory management scheme which supports non-contiguous memory allocation. In this scheme, a process could be allocated in different parts and each part may reside at different part of the memory. In other words a process is divided into several segments and each segment may vary in size. For example, segments of a process may correspond to stack, sqrt function, main program, variables etc. Each segment of a process can be allocated to different memory segments. Thus, a logical address space of the process is a collection of all segments associated with the process. Each segment is identified by its segment number. A logical address generated by CPU in this scheme consists of two parts: segment number and offset.

<Segment number, offset>

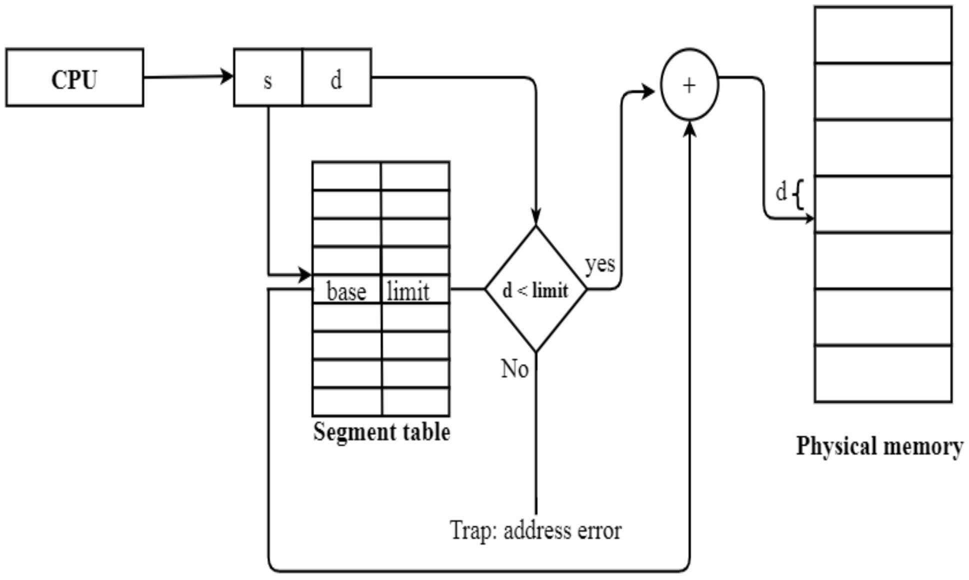


Figure 6.17 : Segmentation hardware [1].

This logical address must be mapped to actual main memory address. This mapping is a provided by a segment table. Each process has its own segment table as shown in Figure 6.17. Every entry in the segment table consists of two fields: a segment base and a segment limit. The segment base specify the starting address of a segment in main memory. While, the segment limit specifies length of the segment. When CPU generates a logical address for a process, it consists of a segment number s and a segment offset d. The segment number is used to index into its segment table. The segment offset must be between 0 and segment limit. If, the segment offset is outside of this range, OS will generates trap. But when the segment offset is legal, it is added to the segment base address to generate the required address in the memory.

As an example, consider a situation given in Figure 6.18. The physical memory contains fives segments numbered from 0 through 4 as shown. The segment table has one entry for each segment containing base address (the beginning address of the segment in physical memory) and limit (the length of that segment). For example, segment 3 is 1100 bytes (or words) long and begins at location 3200. Thus, a reference to word 44 of segment 2 is mapped onto location 3200 +44= 3244. A reference to word 107 of segment 2, is mapped to 4300 (the base of segment 2) + 107 = 4407.

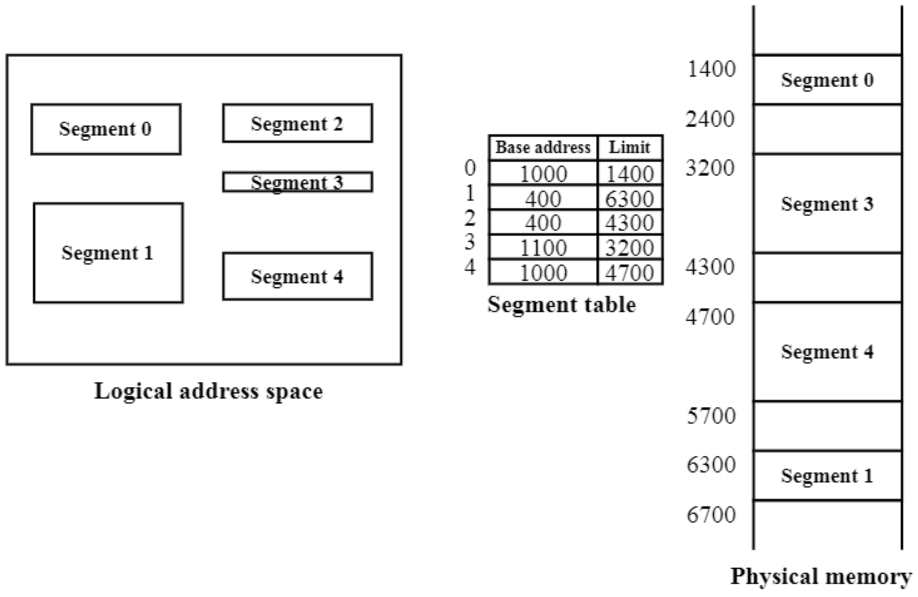


Figure 6.18 - Example of segmentation [1].

Advantage: In segmentation, there is no internal fragmentation. Also, a segments can be shared between processes, and each segment can have some protection info; for example, the code section could be read-only.

Disadvantage : There is an external fragmentation in segmentation.

Illustrative Question : For each of the four processes P1, P2, P3 and P4. The total size in kilobytes (KB) and the number of segments are given below.

Process	Total size (in KB)	Number of segments
P1	195	4
P2	254	5
P3	45	3
P4	364	8

The size of an entry in the segment table is 8 bytes. The maximum size of a segment is 256 KB. What is the size of segment table?

Solution :

Number of segments= 4+5+3+8= 20

Number of bits required to address each segment= 5 bits (since 2^5 = 32 and 2^4=16)

Maximum segment size= Max (195/4,254/5, 45/3,364/8) = Max (49,51,15,46)= 51 bytes.

Number of bits required for segment size or segment offset= 6 bits (since 2^6 = 64 and 2^5=32)

S	D
5 bits	6 bits

Size of segment table=number of segment× segment table entry size= 2^5×8= 256 bytes

Check your progress

1. What is the major problem with segmentation?
2. What are the advantages and disadvantages of inverted page table?
3. What are the advantages of using hierarchical page table over traditional page table?
4. How two level page table is different from inverted page table?

6.10 VIRTUAL MEMORY

Earlier, we have seen that an entire process (all its pages) must be brought into the main memory for its execution. Virtual memory is a technique which facilitate execution of a process which is not completely brought into the memory. With this technique, we can execute a process whose pages are not completely loaded into the main memory. Thus, a process which is even larger than main memory can be executed with this scheme as shown in figure 6.19.

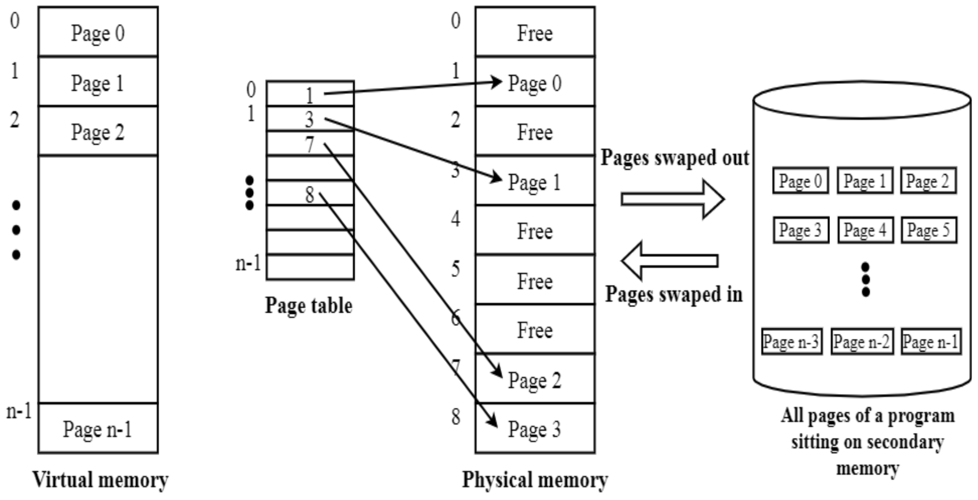


Figure 6.19: Diagram showing virtual memory that is larger than physical memory [1].

It has been shown that an entire process is not needed to load into the main memory for its execution because certain features and part of a process is rarely required. Even if, the entire process or pages of the process are required, but all pages are not required at the same time during its execution. There are many benefits of bringing only required pages during execution:

1. A process or program larger than the main Memory can be executed. So, programmers need not worried about the size of the program. They can write a program even much larger main Memory.

2. Since, the user program could take less physical memory, more programs can be loaded into the memory and it will increase degree of multiprogramming of the system. Thus, the CPU utilization and throughput also get increased.

3. Less I/Os (only for some pages) will be required to swap in or swap out a process.

The virtual memory is generally implemented through demand paging which we will discuss now.

6.11 DEMAND PAGING

Demand paging is commonly used technique to implement virtual memory. In this technique, pages are loaded into the main memory only when they are required during execution. Thus, Pages which are never needed, will never be loaded into the main Memory. Demand paging is similar to paging system with swapping where processes are reside on secondary memory. But in demand paging instead of swapping entire pages of a process, it swaps only those pages into the memory that are currently needed. A demand paging never brings a page into the memory, if it will not be required.

Implementation : When a process needs execution, the system guesses which pages it will be using. Only these pages are brought into the memory. This decreases the swap time and main Memory space. This scheme requires some sort of hardware support to differentiate between those pages that are on memory and those pages that are on the disk. The valid -invalid bit that we have discussed earlier can be used here with little modification. Here, if the bit is set to valid, then the associated page is legal (the page is in logical address space of the process) and is present in the memory. But if this bit is set to invalid, the associated page is either not legal (the page is not in logical address space of a process) or currently on the disk as shown in Figure 6.20.

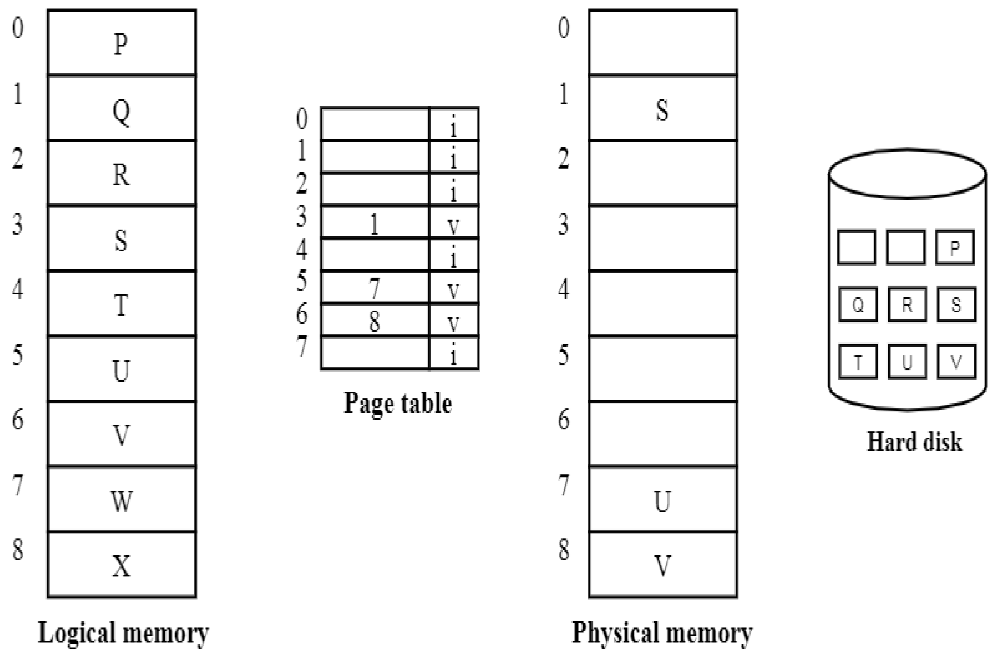


Figure 6.20 : Page table when some pages are not in main memory [1].

If we guess right pages and brings into the memory these pages, the process will execute normally like all pages are brought into the memory. But when OS try to access a page that is not in the memory i.e marked invalid in page table entry, a page fault will be generated. The page fault situation is handled as follow (as shown in Figure 6.21):

1. The requested memory address is first checked, to make sure it was a valid memory request.

2. If there is request for invalid page, the process is terminated. Otherwise, the page must be bring into the physical memory.

3. A disk operation is scheduled to bring in the required page from disk. (This will usually block the process to waiting state and CPU busy with some other process)

4. When the I/O operation is completed, the process's page table is now updated with the new frame number (one of free frame from a free-frame list), and the invalid bit is changed to valid to indicate that the requested page is brought into the memory.

5. The instruction that caused the page fault is restarted from the beginning when this process gets CPU.

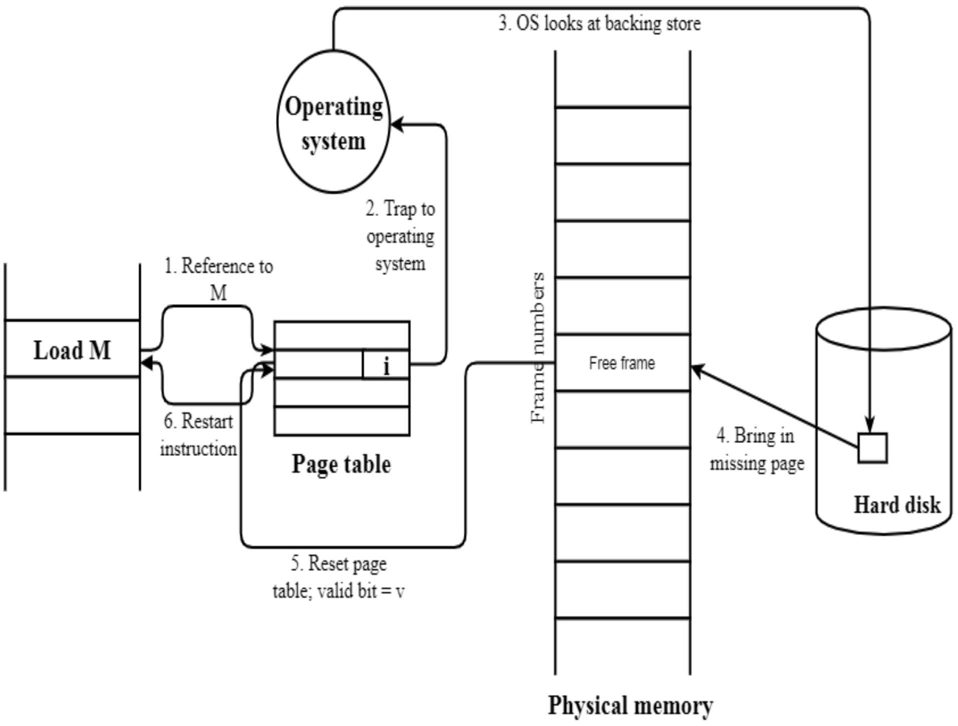


Figure 6.21 : Steps in handling a page fault [1].

In extreme case, a process starts execution with a page fault i.e. no pages loaded into the memory. After the page fault, the page is loaded into the memory. The process continue to fault until all needed pages are brought into the memory. After that, the process continue to run normally with no more page fault. This scheme is pure demand paging. For example, consider a system with 5 process and total 40 frames in memory. The size of each process is 10 pages, however each process is using its only 5 pages and rest 5 pages are unused throughout execution of each process. Under this scenario, the system could run all the processes without any page fault. Demand paging has saved 50% of I/O for each process by not bringing the remaining 5 pages into memory. This way, it increases degree of multiprogramming along with CPU utilization and system throughput.

Illustrative Example : Let the page fault service time be 10ms in a computer with average memory access time being 20ns. If one page fault is generated for every 106 memory accesses, what is the effective access time for the memory?

Let P be the page fault rate

Effective Memory Access Time = p * (page fault service time) +

(1 - p) * (Memory access time)

= (1/106)* 10 * (106) ns +

(1 - 1/106) * 20 ns

= 30 ns (approx)

6.12 PAGE REPLACEMENT

When a user process is executing and at some point of time, it generates a page fault. The OS will then find a free frame. But, OS notices that there is no fee frame available in free frame list and all frame are already occupied. Then, a memory frame must be freed and allocated to the process which fault for the page. This operation is performed by a Page Replacement Algorithm. During a page fault service routine following activities are performed:

1. Find the location of the page on the disk.
2. Find a free memory frame.
   a) If a free memory frame is present in the free frame list, use it.
   b) If there is no free frame in free frame list, use a page replacement algorithm to select a victim frame.
   c) Write the victim frame to the disk and change related values in its page table.
3. Transfer the desired page stored on the disk into the free frame. Change associated values in page table and frame table.
4. Restart the process.

Demand paging requires Page Replacement Algorithm and Frame Allocation Algorithm. Multiprogramming systems have multiple processes in the main memory. We must allocate a certain numbers of frames to each process. This is done by a Frame Allocation Algorithm. While, a Page Replacement Algorithm selects a victim frame during a page fault.

Reducing page fault service time : In above page fault service routine, you have noticed that of there is no free frame, two page transfer (one for swap out and one for swap in) have occurred. This doubles page fault service routine which ultimately increases effective access time. This can be avoided by using a modify (or a dirty bit) in the page take entries which requires a hardware support. When a page is modified during execution of the associated process it's modify bit is set to 1. Whenever a page replacement algorithm selects a page for replacement. Before writing its content on disk, OS checks it's modify bit. If its modify bit is set to 0, then the page has not been modified and there is no need to write it on the disk. But when its modify bit is set to 1, this indicates that the page has been modified. Under this situation, the page must be written to disk before it is replaced by the new page. This technique only applies to read only pages (like binary executables) that could not be modified.

Evaluation criteria : There are many Page Replacement Algorithm, but we generally selects the one which gives minimum page faults. The page replacement algorithms are evaluated on this parameter by running them on a particular string of memory references. This string of memory references also known as reference string. This reference string can be generated artificially with any random references of memory. This can also be generated by tracing a system and records the addresses of memory references. For example, consider the following reference string.

0100,0432,0101,0612,0102,0103,0104,0101,0611,0102,0103,0104,0101,0610,0102,0103,0104,0101,0609,0102,0105

If, pages are 100 bytes per page, above sequence can be reduced to following reference string :

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

To determine the number of page fault associated with a particulate page replacement algorithm, we need to know the number of frames available to it. Generally, as the number of frames available increases the number of page fault also decreases as shown in Figure 6.22.

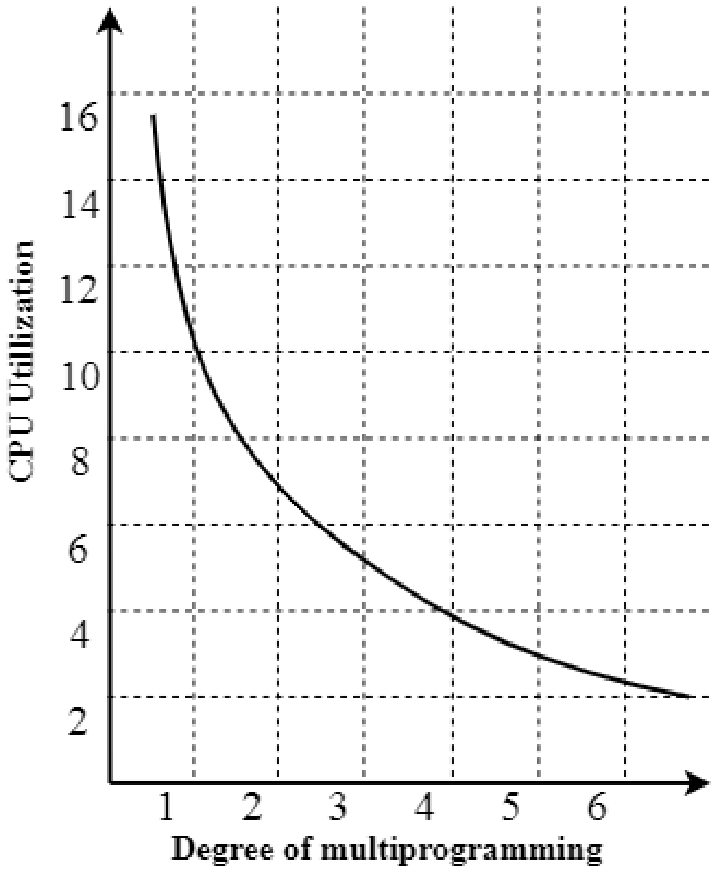


Figure 6.22: Graph of page faults versus number of frames [1].

For the earlier reference string and with 3 available memory frames, we will have three page faults. Now, we will discuss various types of page replacement algorithms along with their evaluation on a particular reference string.

FIFO Page Replacement :

First in First out is simplest page replacement algorithm where the oldest frame is always selected for replacement. FIFO algorithm maintains arrival time for each page and when a page is to be replaced, the oldest page is selected based on the arrival times of pages. The algorithm can be easily implemented through FIFO queue. When a new page enters the memory, it is inserted at the tail of queue. A victim page to be replaced is selected from the head of the queue.

For example, consider the reference string discussed earlier and there are three memory frames available. Initially, the first three references (7, 0, 1) cause page fault and these pages are brought into the available frames. Next, the page reference of 2 causes page fault. Since, the page 7 is brought first into the memory, it is replaced with page 2. The reference for page 0 does not cause any page fault as it is already present in memory. Now, the reference for page 3 replaces page 0 because it is brought first after page 7. Similarly, at any time if a page fault occurs, it is replaced with the page brought first into the memory as compared with other two pages into the memory frame. This process continue for reference to remaining pages and total 15 page fault occurs. The process is shown in figure 6.23 where every time when a page fault occurs the replaced page is shown in one of the three frames.

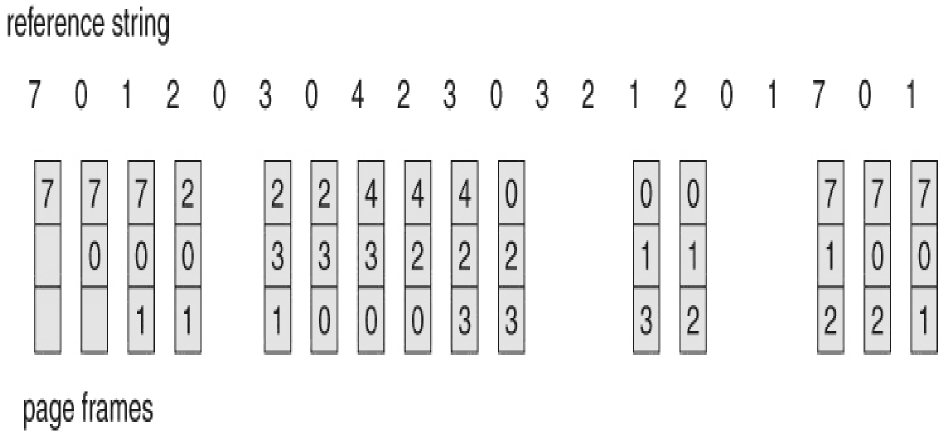


Figure 6.23 : FIFO page-replacement algorithm. [1]

In this page replacement algorithm, everything works perfectly fine until the heavily referenced or the active page is available in memory. But when the active page is replaced with some other pages, a page fault occurs immediately or in near future to bring back the active page into the memory. This happens because of bad page replacement algorithm choice. It is generally obvious that there will be less page fault as we increase the available memory frames. But, this assumption is not always true because of Belady anomaly. It states that for some page replacement algorithm, the number of page fault may increase as we increase the number of memory frames available. This is also shown in figure 6.24.

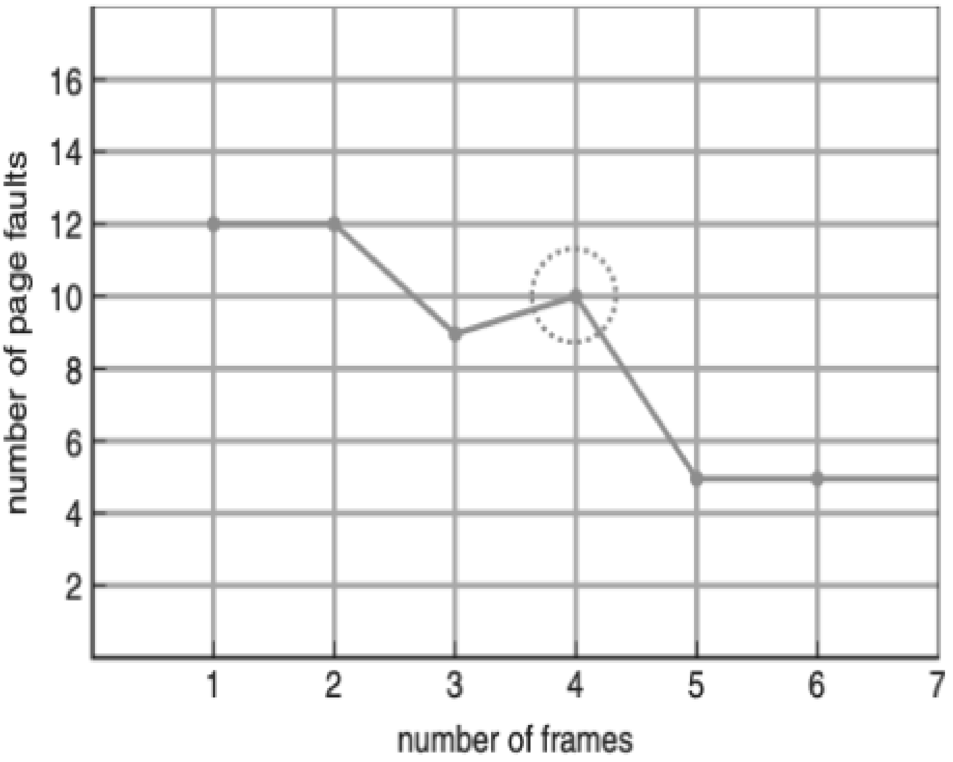


Figure 6.24: Page-fault curve for FIFO replacement on a reference string [1].

Optimal Page Replacement

The optimal page replacement algorithm guarantees minimum page faults and never suffers from Belady anomaly. However, this algorithm practically don't exists and generally used for performance comparison with other algorithms. The optimal page replacement algorithm replaces the page that will not be used for longest period of time.

For example, consider the same reference strings discussed earlier. This algorithm will give a total 9 page faults. As usual the first three references (7, 0, 1) initially don't exist in memory and this causes three page fault. The next reference of page 2 will replace the page 7 since, it will not be used for longest period of time as compared to pages 0 and 1. Now, the reference for page 2 causes page fault and it replaces the page 1. In other words, we can find a victim page 1 by marking in the reference strings, the first reference of the pages (2,0,1) currently available in memory frames after the current reference page and select the page which is marked near the end of the reference strings. As we continue further till the last reference of the page, this algorithm will give total 9 page faults (as shown in figure 6.25) which is twice as good as FIFO replacement algorithm.

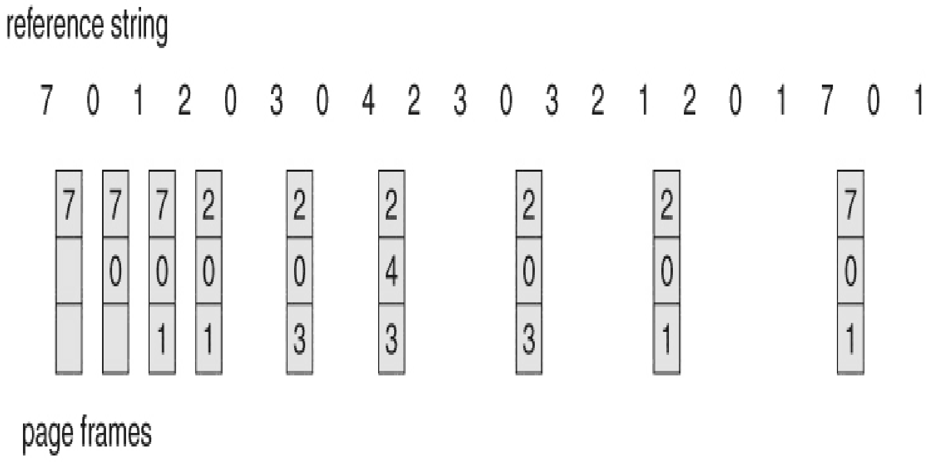


Figure 6.25 : Optimal page-replacement algorithm[1]

LRU Page Replacement

LRU page replacement algorithm selects a page for replacement which has not been used for longest period of time. LRU algorithm maintains last used time of each page. When a page fault occurs, LRU selects the page which has not been used recently. This algorithm is similar to optimal page replacement where we look backward in time for selecting a page for replacement. One strange property of both OPT and LRU replacement algorithm is that the number of page faults on a given reference strings S is same as the number of page faults in reverse of the reference studying Sr.

For example, consider the same earlier reference string. The first five page faults for LRU is same as the optimal page replacement algorithm which is shown in figure 6.26.

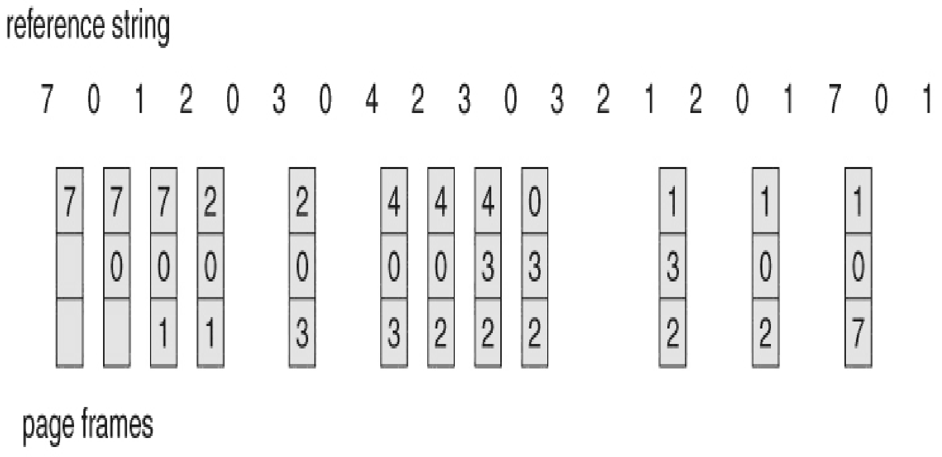


Figure 6.26 : LRU page-replacement algorithm[1]

The 6th page fault occurs when reference for page 4 in the reference string occurs. At this time, LRU algorithm will select page 2 for replacement.

We can find the page to be replaced by marking last reference of each pages (2, 0, 3) present in memory frame before the current reference string (page 4). The page which is marked near the beginning in reference string is then selected for replacement. When the algorithm is run on whole reference string, it will give total 12 page faults. The result is much better than FIFO replacement algorithm. The LRU page replacement algorithm is considered as a good replacement algorithm and often used in operating systems. Like optimal page replacement, LRU also never suffers from Belady anomaly. LRU is implemented in two ways:

Counter : In counter implementation of LRU page replacement, a page table consists an extra entry which contain time-of-use field. A logical clock or counter is added to CPU which is incremented for every page reference. Whenever a reference for a page occurs, the content of the logical clock is copied to the time-of-use field of the page in the page table. This way, the page table maintains the last references of each pages. Wherever a page fault occurs and a page need to be selected for replacement, entire entries in the page table is searched to find the page with smallest time value. So, this implementation requires search of entire page table.

Stack : In this implementation of LRU, a stack of page numbers is maintained. Whenever a reference for a page occurs, the page number is pushed into the stack. If the page number is not present in the stack, it is simply pushed at the top of the stack. Otherwise if the page number already exists in the stack, it is first removed from the stack and pushed at the top of the stack (as shown in figure 6.27).

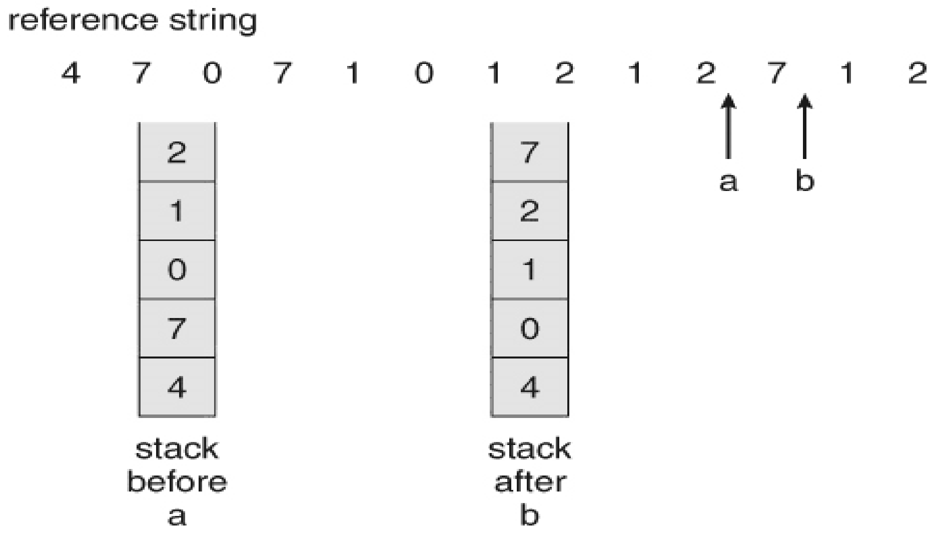


Figure 6.27 : Use of a stack to record the most recent page references[1]

The stack update operation is a little expensive but it do not requires any search like the counter implementation. The stack is generally implemented through doubly linked list with a head and tail pointer.

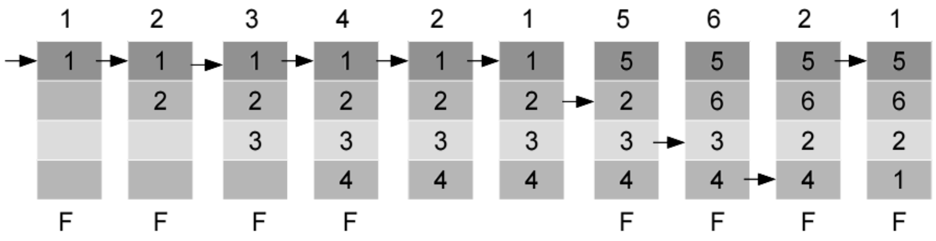
Illustrative Questions: Given page reference string :

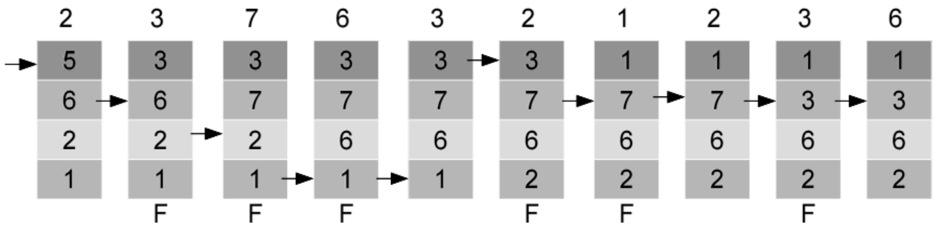
1, 2, 3, 4, 2, 1, 5, 6, 2, 1, 2, 3, 7, 6, 3, 2, 1, 2, 3, 6

Assume that there are 4 page frames which are initially empty. Compare the number of page faults for LRU, FIFO and Optimal page replacement algorithm.

Solution :

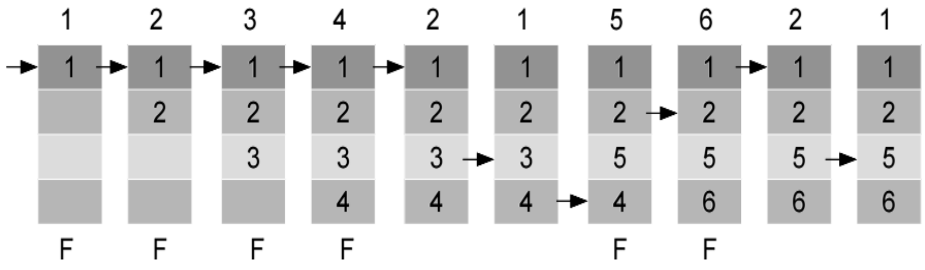
First In First Out : The arrow pointing to the memory frame is first came (among others currently in memory) which is to be replaced in case of page fault at particular page request.

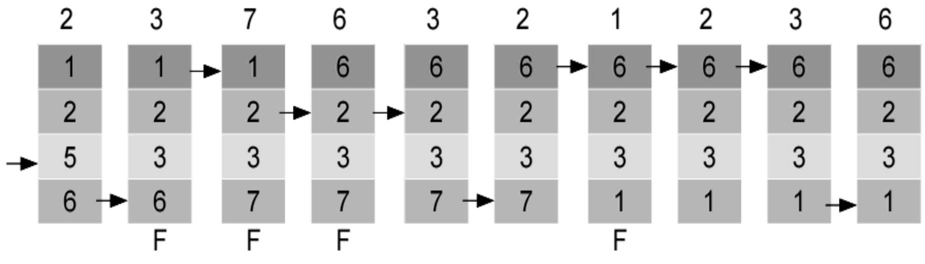




Total page faults = 14

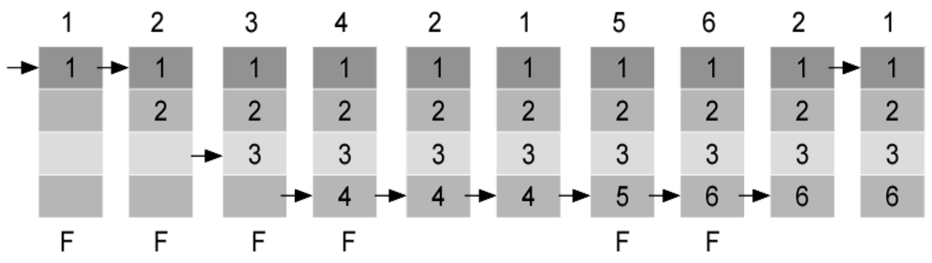
Least Recently Used: The arrow pointing to the memory frame is Least Recently Used frame which is to be replaced in case of page fault at a particular page request.

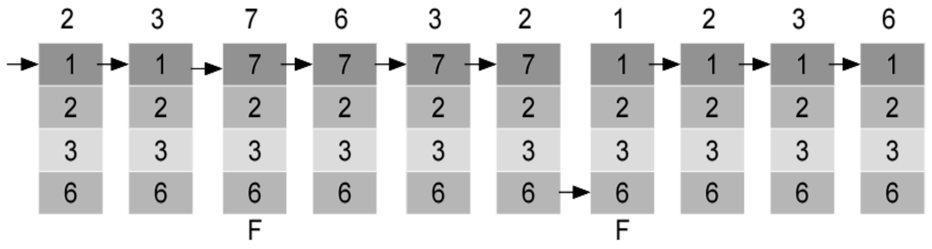




Total page faults = 10

Optimal Page replacement : The arrow pointing to the memory frame is the page that will not be used for longest time as compared to other pages currently in memory frame and it is to be replaced in case of page fault at a particular page request.





Total page faults = 8

Check your progress

1. Assume that there are 3 page frames which are initially empty. If the page reference string is 1, 2, 3, 4, 2, 1, 5, 3, 2, 4, 6, what is the number of page faults using the optimal replacement policy?
2. Consider a main memory with five page frames and the following sequence of page references: 3, 8, 2, 3, 9, 1, 6, 3, 8, 9, 3, 6, 2, 1, 3. Which one of the following is true with respect to page replacement policies First-In-First Out (FIFO) and Least Recently Used (LRU)?

   (A) Both incur the same number of page faults

   (B) FIFO incurs 2 more page faults than LRU

   (C) LRU incurs 2 more page faults than FIFO

   (D) FIFO incurs 1 more page faults than LRU

6.13 ALLOCATION OF FRAMES

One of the major concern in virtual memory environment is how do we allocate a fixed available memory to various processes. For example, if we have 100 free frames and the system has only two processes, how many frames each process will get? One basic strategy is to consider a single user system. Assume the system supports 128K memory with page size of 1K. If the OS needs 28 memory frames, we will left with 100 free frames for use processes. Under pure demand paging, when processes starts their execution, they generate page faults. After 100 page faults, all free frames will be allocated to these processes. When the free frame list will be exhausted, a page replacement algorithm will replace one of the page on the existing frames with new page to handle 101 page fault. This continue to further page faults. At the end when all process finish their execution, all free frames are added back to free frame list.

The above basic strategy requires some considerations. We cannot allocates more than available free frames to process. Also, a process must get minimum number of frames to hold different pages of any instruction. For example, one level indirect addressing like load instruction on page 15 may reference to an indirect address on page 11 which may reference page 9. So this instruction requires at least 3 pages to support paging. So the minimum number of frames to be allocated to each process depends on system architecture. While the maximum number of frames to be allocated for each process depends on available physical memory. As the number of frames allocated to each processes decreases, the number of page faults increases.

Allocation Algorithms :

Equal Allocation - The simplest way to allocate free frames among user processes is to distribute m/n frames to each process. Here, m is the total number of free frames and n is the total number of processes. This scheme is known as equal allocation.

Minimum number of frames per process= total number of free frames (m)/ total number of processes (n)

Proportional Allocation – Since, each process requires different memory needs, it is not a good choice to allocate same number of frames to each process. For example, consider a system with 1K words page size. The system has a student process of 20K words and a printer process of 150K words. So the student process requires 20 frames while printer process requires 150 frames. If the system currently has 200 free frames, it do not make sense to allocate 100 frames to each process because the student process requires only 20 frames. This problem is solved by Proportional Allocation where each process are allocated frames according to their sizes. Each process of size si and m free frames will be allocated si/S * m frames. Where S is total size of all processes.

Number of frames to be allocated (ai)= size of process (si)/ total process size (S) * number of free frames (m)

With Proportional Allocation the student process will get 24 frames =20/170 * 200. While the printer process will get 177 frames=150/170*200. This way, frames requirements of each process can be satisfied.

Global versus Local Allocation

Another important factor in allocation of frames is Global and Local page replacement. The number of frames allocated to a process changes dynamically during its execution as page faults occur. This depends on Global and Local Page Replacement.

Global Replacement : Global Replacement allows a process during page faults to take frames from list of all frames even the frames are currently allocated to other process. In other words, when page faults occurs, a process can take frames from other processes along with frames allocated to itself.

Advantage : It does not hinder a process because the process can take frames containing less used pages of other process. This increases the system throughput.

Disadvantage : A process cannot control its own page fault rate because its paging behaviour is also depend on execution of other processes.

Local Replacement : With local Page Replacement, the number of frames allocated to a process is fixed and does not changes during its execution. Local Replacement allows a process to take frames only from its own set of allocated frames. The process cannot take frames allocated to other process.

Advantage : The paging behaviour of a process only dependant on only its own execution.

Disadvantage : A low priority process might hinder the performance of a high priority process by not allowing a high priority process to take frames allocated to the low priority process.

6.14 THRASHING

When a process does not get minimum number of frames allocated to it as defined by computer architecture, it cannot continue its execution. If, the process does not has sufficient frames to support its pages in active use, it will immediately cause page fault. Since all the pages are in active use, one of the active pages that may be needed next must be replaced to serve the page fault. As the result, the process will quickly fault again and again. In this scenario, the process spends more time in doing swap in and swap out rather than its execution. This high paging activity where a process spends more time in doing swapping and there is no progress in the process's execution is called Thrashing. Thrashing results in serious performance degradation of a system. It causes no progress in processes execution and the system throughout decreases significantly.

Causes of thrashing : Consider a situation where Operating System continuously monitors a system performance. If, the CPU utilisation decreases, OS increases degree of multiprogramming by introducing new processes. Since, a global page replacement is used, when the new processes needs more frames, it starts page faults. So the process will take away the frames from other processes. But other processes need these frames, so they also starts page faults. These processes will need paging device for swap in and swap out. So, they queue up long for paging device. The CPU utilisation decreases and the ready queue becomes empty. When OS (more precisely CPU scheduler) sees decrease in CPU utilisation, it increases the degree of multiprogramming and the same process continue as described above. As the result, thrashing occurs and page fault rate increases significantly. Ultimately, the system throughput drops tremendously. This phenomenon is shown in figure 6.28.

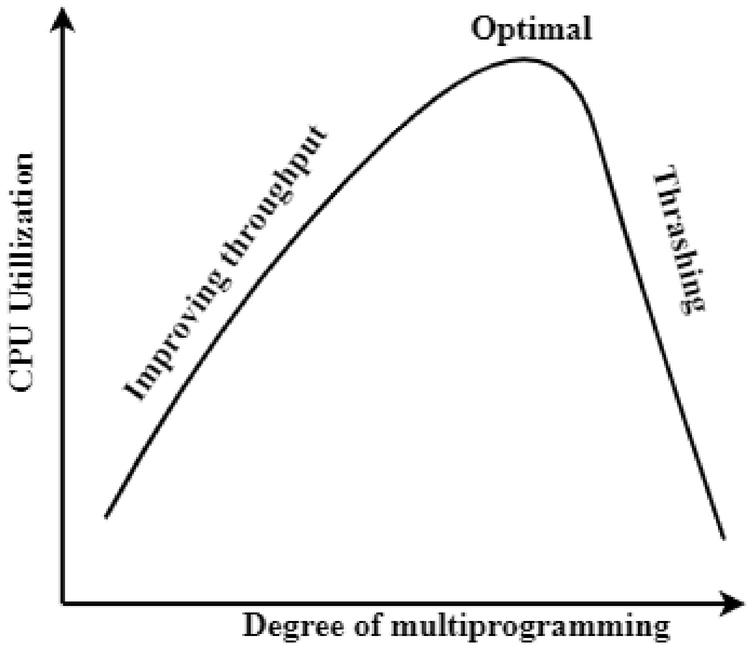


Figure 6.28: Thrashing [1].

As the degree of multiprogramming increases, the CPU utilisation also increases slowly until a maximum is reached. If the degree of multiprogramming is further increased beyond this point thrashing occurred and CPU utilisation stars decreasing.

We can limit the thrashing by using local page replacement. The thrashing process will not take frames from other processes to cause these processes to thrash as well. But, this will not solve the problem completely. Because the thrashing processes increase average page fault service time (since the waiting queue for paging device has increased). Next we will discuss the Working Set Model and Page fault frequency to prevent thrashing.

Prevention from thrashing :

Working-Set Model :

The Working Set Model to prevent thrashing is based on locality of reference. Locality of reference states that a same set of memory location or pages will be repeatedly used again in near future. This model uses a Δ parameter (Working set window size) to define a working set (set of most recently used pages in Δ page references) for each process. For example, if Δ =10, then a working set of a process at time t1 is {1, 2, 5, 6, 7} and its working size at time t2 is {3, 4} as shown in figure 6.29.

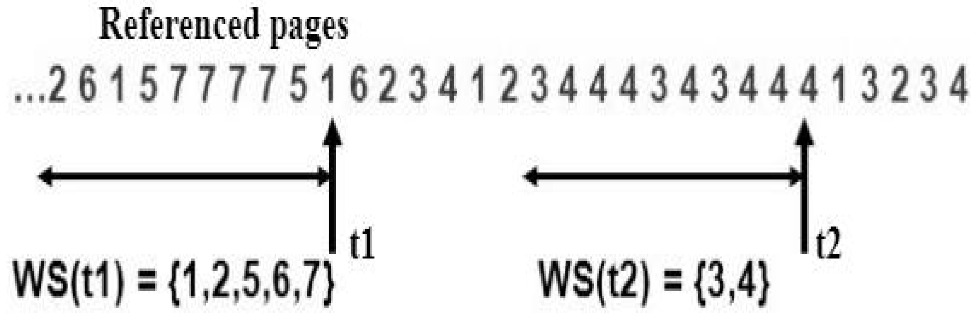


Figure 6.29 : Working-set model [1].

While considering a working set of a process at any time, we only takes unique page references in window size of 10 page references. After knowing the working set window and working set for each process, we will compute total demands of frames D. This can be computed by calculating working set size WSSi for each process.

D=sum (WSSi)

Here D is total demands for frames and WSSi is number of frames needed by process i. If the total demands for frames D is greater than total number of available frames m, then thrashing will occur. The OS continuously monitors working set of a process and allocates frames according to its working set size. If there are enough extra frames left, a new process can be initiated. Otherwise, if total demands of frames exceeds total number of available frames in free frame list, OS selects one process to suspend. The frames allocated to the terminated process are then allocated among other processes. The suspended process will be restarted later.

Page Fault Frequency : Another way to prevent thrashing is Page Fault Frequency which is more direct way. We have seen earlier that during thrashing page fault rate increases significantly. So if we can control the page fault rate, we can ultimately control thrashing. The page fault frequency does the same thing. When the page fault rate increases this means a process need more frames and when the page fault rate decreases a process has too many useless frames. Page fault frequency method set an upper bound and lower bound on page fault rate as shown in figure 6.30. If the page fault rate exceeds the upper bound, frames are allocated to the processes. Otherwise, if the page fault rate drops below the lower bound, frames are removed from processes.

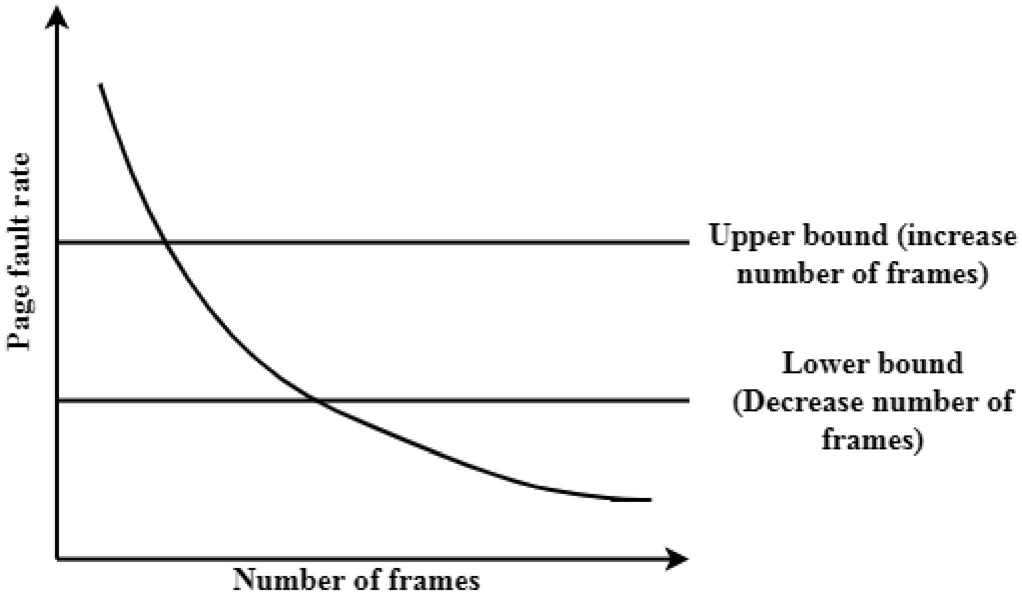


Figure 6.30 : Page-fault frequency [1].

6.15 SUMMARY

In summary, we discussed:

- You understand the concept of virtual memory which is commonly implemented through the demand paging.
- We discussed various types of page replacement algorithm to serve page faults when frames in free frame list are exhausted.
- We discussed equal and proportional allocation of frames to processes. We have also seen Global and Local Page replacement which control allocation of frames to different processes.
- You understand what is thrashing and how does it occurs. We have seen how do we prevent thrashing.

6.16 TERMINAL QUESTIONS

1. What are Techniques to prevent thrashing.
2. How does thrashing occurs?
3. How does Working at model prevent thrashing?
4. How does Page Fault Frequency control thrashing?
5. What are the consequences of Thrashing?

6. What is the benefit of Proportional Allocation over equal Allocation?

7. What are the advantages and disadvantages of Global Page replacement over Local Page Replacement?

8. What is hierarchical paging?

9. Explain how does internal fragmentation is possible in paging.

10. Describe how does the mapping of logical address space to physical address space happens.

11. Explain following:

    a) Page table

    b) Frame table

    c) Segment table

12. Explain how does logical addresses of a process is translated to physical addresses in segmentation.

13. Which page replacement policy sometimes leads to more page faults when size of memory is increased?

14. Whether following statements are true or false:

    a) The amount of virtual memory available is limited by the availability of the secondary memory.

    b) The best fit techniques for memory allocation ensures that memory will never be fragmented.

UNIT-VII SECONDARY MEMORY MANAGEMENT

Structure

7.1 INTRODUCTION

Secondary and tertiary storage structures are the lowest level of the file system. Magnet disks like hard disks, floppy disks are most common type of disks. Optical disks (like CD-ROM, DVD) are common for distribution of movies, music, data and programs. In this chapter, we will first look into the physical structure of magnetic disks and various parameters on which their performance depends. Next, we will see how data are stored on disks. Then we will discuss how disk I/Os requests are served to improve its performance and uses it efficiently. Later we will discuss free space management to keep track of free disk space. Finally, we will explain how the kernel manage swap space located on disk.

7.2 OBJECTIVES

After studying this chapter, you should be able to understand:

- Brief review of how Magnetic disks work and how data are organized on to disks.
- How to schedule the order of disk I/Os requests to improve performance.
- How OS keeps track of free disk space.

- How does the swap space work in the virtual memory system?

7.3 MAGNETIC DISK STRUCTURE

Magnetic disk offers a huge amount of secondary storage to modern computer systems. Magnetic disk consists of one or more platters (a flat circular shape, like a CD) coated with magnetic material. Each platter has two working surfaces where we store data. Data are recorded magnetically on each platters. Each surface contains a number of concentric rings called tracks. All tracks immediately above one another (as shown in figure 7.1) form a cylinder. A typical magnetic disk has thousands of concentric cylinders. Each track further contains sectors which can store traditionally 512 bytes of data each. There are read-write heads that read data stored on sectors. The standard configuration have one head for each surface. There is a common arm assembly that controls all heads and move them simultaneously from one cylinder to another.

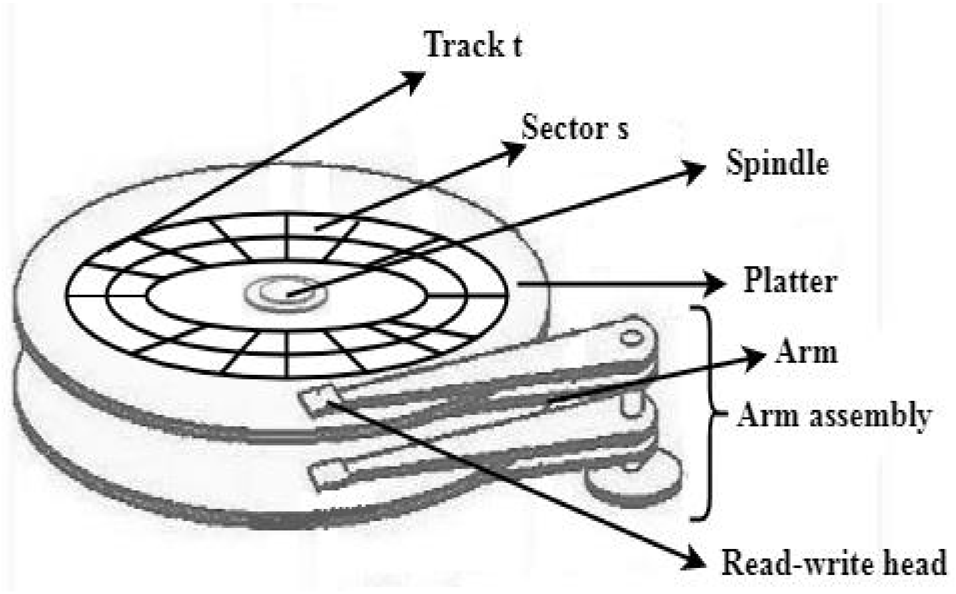


Figure 7.1 : Moving-head disk mechanism [1].

When we are using the disk, a drive motor attached to the spindle spins each platter simultaneously with high speed at around 60 to 200 rotation per second. Transfer of data between disk and computer happens as the disk rotates. A rate at which data flows between the magnetic disk and the computer is called transfer rate. The transfer rate depends on two factors: rotating speed of the disk and number of bytes to be transferred. The positioning time which is also called as random access time is the time required to move the disk head to desired sector. This positioning time depends on two parameters: seek time and rotational latency. The seek time is the time taken by the disk arm to move to the desired track. While, rotational latency is the time taken to rotate the desired sector to reach over the disk head. The disk arm moves the disk head to correct track and waits for the disk to rotate so that the head comes over desired sector.

Transfer of data stored in the sector happens as the disk spins. Generally, a disk transfers data at several megabytes per second. The seek time and rotational latency of a disk are several milliseconds. The heads that read data from the disk drive are essentially coils of wire. The disk head flies at a small very distance (measured in micron) over the disk platter to read or write data. Even the disk platters are coated with protective later, the head may damage the platters which lead to head crash. In this case many times, we have to replace the entire disk. The heads sense a change in the magnetic field, when they pass through the platters coated with magnetic material. As the result, a current flows through the coils in the disk heads. The 1 and 0 are encoded as north or south pole. The heads sense transitions from one pole to another pole which cause current to flow inside the heads. Areas encoded with multiple 1s and 0s do not cause current to flow through the heads and so these bits are not sensed by the heads.

A magnetic disk may be removable like floppy disk which allows different disks to be inserted onto the disk drive. A Removable magnetic disk consists of only one platter covered with a plastic case to protect the platter when it is not in the disk drive.

Disk Structure : In modern disk drives, a logical block (usually 512 bytes) is the smallest unit of transfer of data. A one dimensional array of logical blocks is mapped on sectors of disk sequentially. The mapping starts with sector 0 located at the first track of the outermost cylinder. Then it proceeds from rest sectors on the rest of the tracks on the same cylinder. The mapping proceeds from outermost cylinder to innermost cylinder. This way, we can convert a logical block number to a disk address containing three fields: a cylinder number, a track number in that cylinder, and a sector number in that track.

Illustrative Example : Consider a disk with the following characteristics: sector size B=512 bytes, number of sectors per track=20, number of tracks per surface=400. number of double sided surface= 15.

(a) What is the total capacity of a track and how many cylinders are there?

(b) What is the total capacity a cylinder and the total capacity of a disk pack?

(c) Suppose the disk drive rotates the disk pack at a speed of 2400 rpm (revolutions per minute); what is the transfer rate in bytes/msec, a block transfer time btt in msec and average rotational delay?

(d) Suppose the average seek time is 30 msec. How much time does it take (on the average) in msec to locate and transfer a single block given its block address? (e) Calculate the average time it would take to transfer 20 random blocks and compare it with the time it would take to transfer 20 consecutive blocks.

Solutions :

(a) Total track size = 20 * (512) = 10240 bytes = 10.24 Kbytes
Number of cylinders = number of tracks = 400

(b) Total cylinder capacity of a cylinder = 15 * 2 * 20 * 512 = 307200 bytes = 307.2 Kbytes
Total capacity of the disk = 15 * 2 * 20 * 512*400 = 122.88 Mbytes

(c) Transfer rate (tr) = (size of a track in bytes)/(time for one disk revolution in msec)

= (10240) /((60 * 1000) / (2400))= (10240) / (25) = 409.6 bytes/msec
block transfer time (btt) = B / tr = 512 / 409.6 = 1.25 msec

average rotational delay (rd) = (time for one disk revolution in msec) / 2 = 25 / 2 = 12.5 msec

(d) Average time to locate and transfer a block = s+rd+btt = 30+12.5+1.25 = 43.75 msec

(e) Disk Access Time = Seek Time + Rotational Latency + Transfer Time

Time to transfer 20 random blocks = 20 * (s + rd + btt) = 20 * 43.75 = 875 msec

Time to transfer 20 consecutive blocks using double buffering = s + rd + 20*btt = 30 + 12.5 + (20*1.25) = 67.5 msec.

7.4 DISK FORMATTING

Before a new magnetic disk can be used to store data, its platter should be divided into sectors to indicate beginning and ending of each sector. This is called low level formatting or physical formatting. During this process, a small data structure is stored on each sector. This data structure consists of a header, a trailer and a data area (usually 512 bytes). The header and trailer contain sector number and Error Correcting Code (ECC). These information allow disk controller to not only detect errors in a sector but also allow it to fix the errors in many cases. When the controller writes data on a sector, it also writes ECC code that was calculated based on all bytes of the data. In future, when this sector is read by the disk controller, it recalculate ECC code and compared with the one stored on the sector. If the calculated value of ECC is different from stored one, then the sector is corrupted. The ECC code contains enough information to recover the data, if only few bits are corrupted.

After the low level formatting of a disk, operating system must put its data structure on the disk before the disk can be used to store files. Storing the data structure require two steps: partitioning of the disk and creation of file system on the disk. Partitioning of a disk is making groups of one or more disk cylinders. For example, one portion can be used to store OS executable files while other partition can be used to store user's files. The operating system treats each partition as a separate disk. After the disk partitioning, OS need to put an initial file system on the disk. The file system maps free spaces, allocated spaces and the root directory of the file system.

Check your progress

1. How does the disk seek time different from rotational latency?
2. How data is organised on the disk?
3. What are tracks and sectors in a disk?

7.5 DISK SCHEDULING

The time taken to read or write disk blocks depends on three factors:

1. **Seek time :** It is the time for the disk arm to move the heads to the desired cylinder.
2. **Rotational latency :** It is the time required to rotate the disk heads to the desired sector.
3. **Disk bandwidth :** It is the total number of bytes transferred, divided by the total time between the first request of transfer to the completion of the last transfer request.

Operating system is responsible for minimizing all above factors to reduce the overall disk blocks read/write time. OS improves seek time, rotational latency and the bandwidth by servicing the disk I/O requests in effective order. A request for disk block is serviced immediately, when the desired disk drive and disk controller are available. If the drive or controller is busy, further new requests are placed in the queue of pending requests for that drive. In a multiprogramming environment with multiple processes, the disk queue often has several pending requests. OS uses a disk scheduling algorithm to process these pending requests in an effective order which minimizes seek time, rotational latency and maximize the disk bandwidth. There are several algorithm exist to service disk I/O block requests which you will see now.

FCFS : First Come First Serve Disk Scheduling algorithm is simplest to all other Disk Scheduling algorithms. FCFS serves disk requests in sequential order i.e in the order they arrive in the disk queue. Consider an imaginary disk with 200 cylinders and a disk queue with requests for I/0 blocks on cylinders: 95, 180, 34, 119, 11, 123, 62, 64. Assume that the Read-write head is initially at cylinder 50 and the tail cylinder is199.

Queue : 98, 183, 37, 122, 14, 124, 65, 67

Head starts at 53

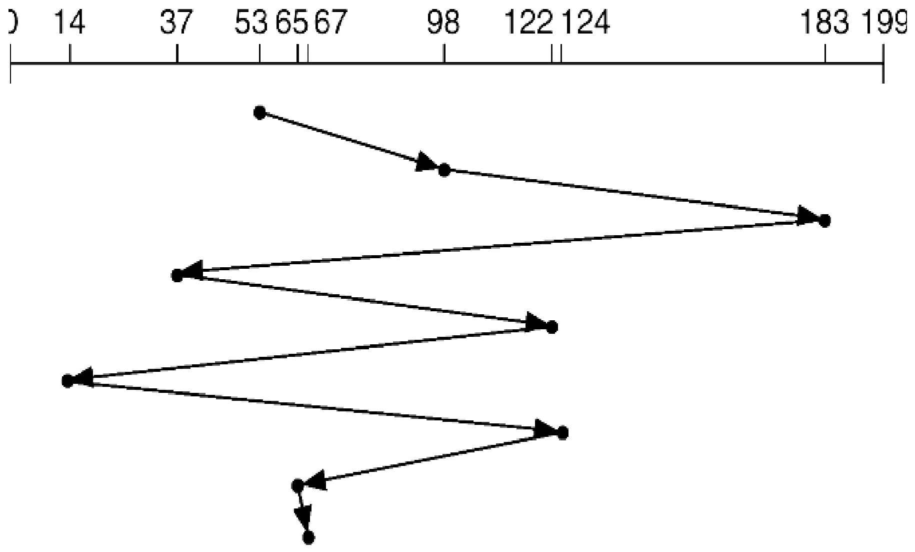


Figure 7.2 : FCFS disk scheduling.

The disk head will starts with cylinder 53 and will move towards 98 to serve it. After serving cylinder 98, it will then serves 183,37,122,14,124,65 and finally it will serves cylinder 67 as shown in figure 7.2. This will cause a total head movement of 640 cylinders.

Total head movement= (98-53)+(183-98)+(183-37)+(122-37)+(122-14)+(124-14)+(124-65)+(67-65)

If you will see a closer look in above example, you will notice that while servicing cylinder 122 from 14 and then back to cylinder 124, causes head movement of 218 cylinders (122-14 + 124-14). This number of head movement could be reduced if cylinders 37 and 14 could be serviced together before or after servicing the requests for cylinders 122 and 124. So, the total head movement could further be reduced which could improve the disk performance.

SSTF Scheduling

In FCFSs you see that it is reasonable to service a request close to current disk head position compared to one which far away from current position. The SSTF services requests that are closest to current disk head position at any time. In other words we can say that SSTF always selects a request which has shortest seek time from current head position compared to all other requests.

Let us consider the same above example. The disk head is initially at cylinder 53. This algorithm will first service the cylinder 65 which is closest to current head position 53. Once the disk head is at cylinder 65, the closest cylinder is now 67. So it is served. This way continuing further, it will service cylinder 37 and then 14, 98,122,124 and finally 183 will be serviced as shown in figure 7.3. In this case there is a total number of head movement of 236 cylinders which is significantly lower than the one in fcfs (640).

Total disk head movement=(65-53)+(67-65)+(67-37)+(37-14)+(98-14)+(122-98)+(124-122)

=236 cylinders

Queue: 98, 183, 37, 122, 14, 124, 65, 67

Head starts at 53

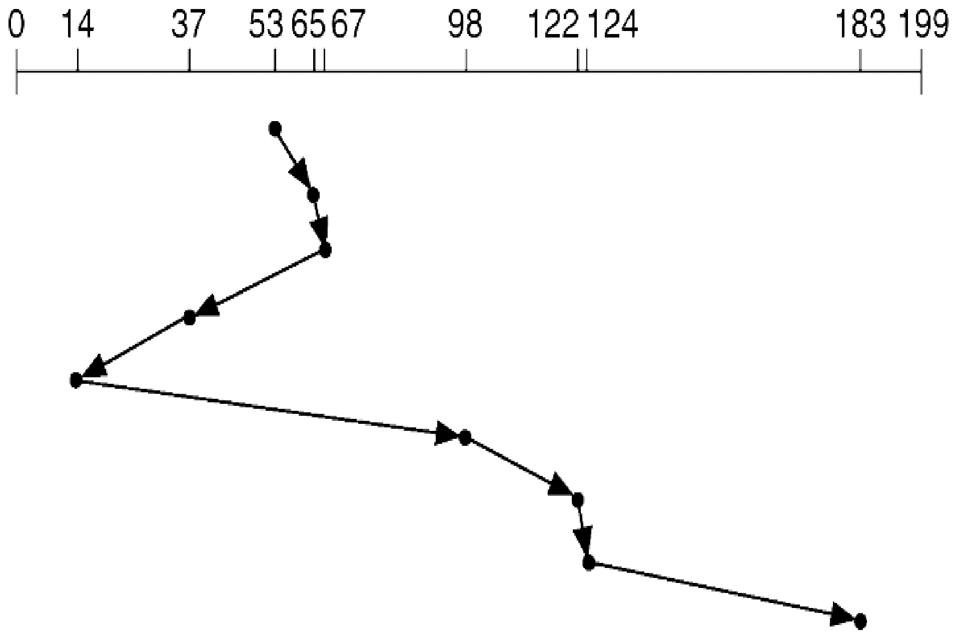


Figure 7.3 : SSTF disk scheduling

Although SSTF algorithm substantial improves the disk performance, it suffers from starvation of some requests. In a multiprogramming environment, requests may arrive at any time. Assume that we have two requests for cylinders 37 and 122 in disk queue. While servicing the request 37, another request for cylinder 14 arrives. So request 14 is now serviced. While servicing the request 14, a new request for cylinder 67 arrives. This causes the request 122 to wait further. If the new requests will always be lower than 122, this would cause the request for cylinder 122 to wait forever indefinitely.

An improvement can be further added to SSTF algorithm to reduce the disk head movement. In the example, we can do better by moving the head from 53 to 37 instead of moving to its closest cylinder 65. After that cylinder 14 should be serviced and moving towards the end of the disk while servicing the coming requests in the way. This strategy would lead to total head movement of 208 cylinders, which we will see next.

Illustrative Example : Consider a disk system with 100 cylinders. The requests to access the cylinders occur in following sequence: 4, 34, 10, 7, 19, 73, 2, 15, 6, 20.

Assuming that the head is currently at cylinder 50, what is the time taken to satisfy all requests if it takes 1ms to move from one cylinder to adjacent one and shortest seek time first policy is used?

Answer :

Since the cylinders are accessed using shortest seek time first scheduling algorithm, the head will first move to 34 from its current position 50. This would takes (50-34)*1 ms. After serving 34, head will move to 20 which would take another (34-20)*1 ms. The head will proceed further so on.

So cylinders will be accessed in following order:

34, 20, 19, 15, 10, 7, 6, 4, 2, 73.

total time taken=(50-34)*1+(34-20)*1+(20-19)*1+(19-15)*1+(15-0)*1+(10-7)*1+(7-6)*1+(6-4)*1+(4-2)*1+(73-2)*1

= 119 ms.

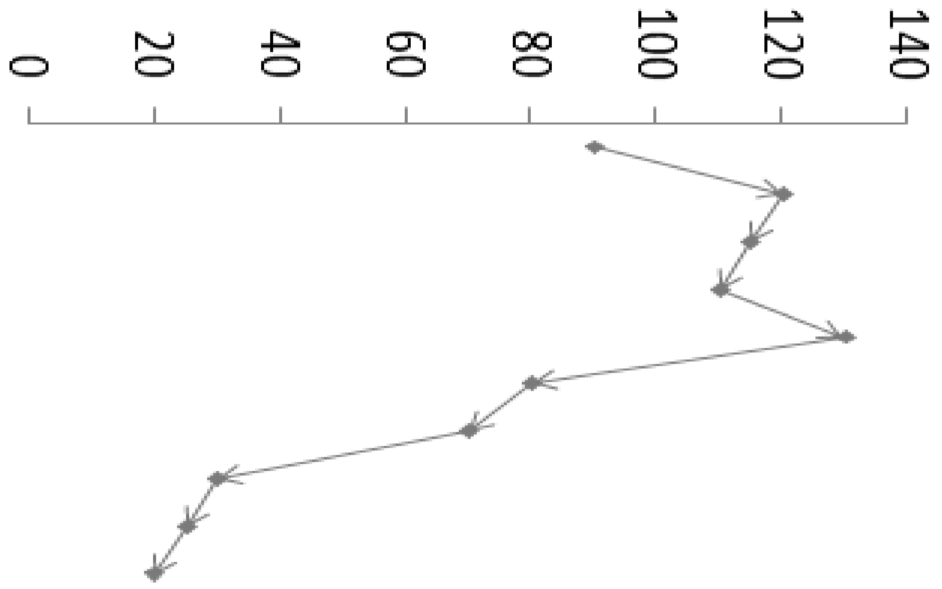
Illustrative Example : A disk has 200 tracks (numbered 0 through 199). At a given time, it was servicing the request of reading data from track 120, and at the previous request, service was for track 90. The pending requests (in order of their arrival) are for track numbers: 30, 70, 115, 130, 110, 80, 20, 25.

How many times will the head change its direction for the disk scheduling policies SSTF (Shortest Seek Time First) and FCFS (First Come Fist Serve)

Answer :

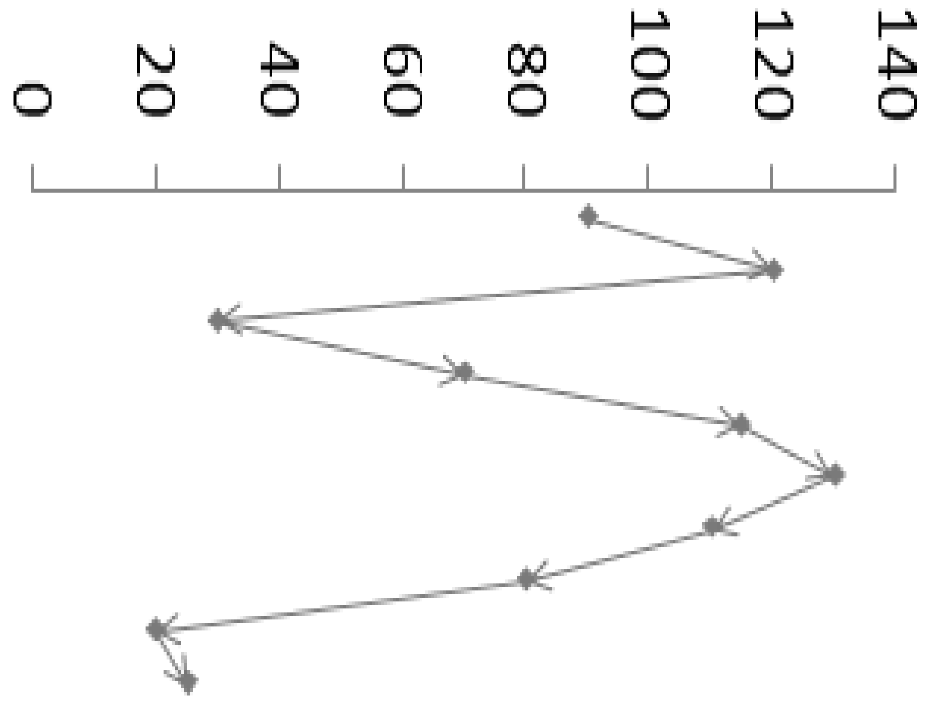
SSTF :

order of servicing tracks: 90, 120, 115, 110, 130, 80, 70, 30, 25, 20



Change of direction are : 120–>115; 110–>130; 130–>80 total direction change=3

First Come First Serve : order of servicing tracks : 90, 120, 30, 70, 115, 130, 110, 80, 20, 25



Change of direction are : 120->30; 30->70; 130->110;20->25

Total direction change=4

SCAN Scheduling SCAN algorithm moves disk head from current position to one end of disk while servicing the coming requests for cylinder in the way. At arriving end of the disk, the head movement direction reversed and continue servicing the upcoming requests in that direction. In this way the head continuously moving towards end of disk while serving the requests in the way just like an elevator in a building. So the algorithm is also called as elevator algorithm.

Considering the above example, SCAN algorithm first finds the direction of disk head movement in addition to its current position. Assume that the head is moving towards cylinder request 0. The disk head would first services the request 37 from its current position 53 as shown in figure 7.4. After that request for the cylinder 14 is serviced and then the head arrives at cylinder 0 which is the one end of disk. At this point, the direction of the heart movement is reversed and then it services the requests 65, 67, 98,122, and finally 183 (as shown in figure) while moving the other end of the disk. This results in total head movement of cylinders 236. We saw that a request arrives in front of the head would be served immediately.

While a request arrives behind the disk head would have to wait until the head reaches the end of disk and reverses the head movement direction.

Queue: 98, 183, 37, 122, 14, 124, 65, 67

Head starts at 53

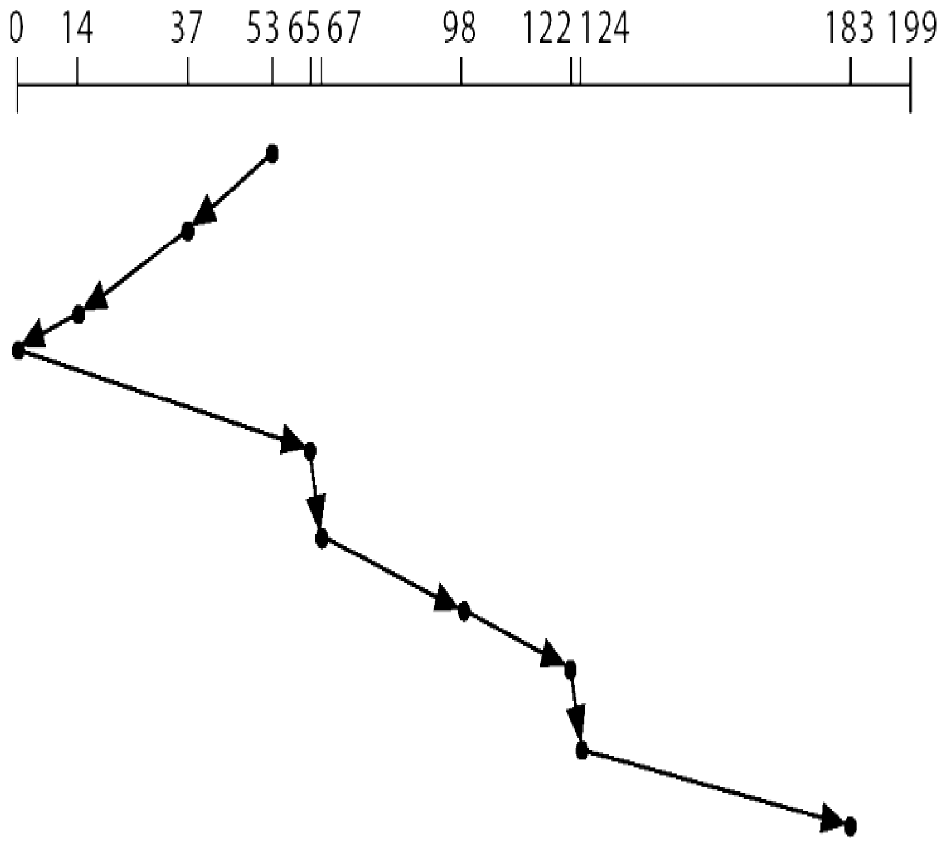


Figure 7.4 : SCAN disk scheduling

Total head movement= (53-37)+(37-14)+(14-0)+(65-0)+(67-65)+(98-67)+(122-98)+(124-122)+(183-124)=236

In SCAN algorithm, when the disk head reaches at the end of the disk and changes its movement direction, at this point there are relatively fewer number of requests in front on the disk head since these requests are already served recently. While there are heavy density of requests near other end of the disk. But these heavy density requests have to wait for longer time than the requests in front of the disk. So, it would be wise to first service the heavy density requests than the requests in front of the head. This strategy is used in the next algorithm.

C-SCAN Circular SCAN disk scheduling algorithm is a variant of SCAN algorithm. Like SCAN algorithm, Circular SCAN algorithm moves the disk head from one end of the disk to other end while servicing the coming requests for cylinder in the way. But, when the disk head arrives at end of the disk, it reverses the direction of head movement and immediately moves to other end of the disk without servicing any requests in the way as shown in figure 7.5. After reaching other end of the disk, it starts servicing the requests.

Queue : 98, 183, 37, 122, 14, 124, 65, 67

Head starts at 53

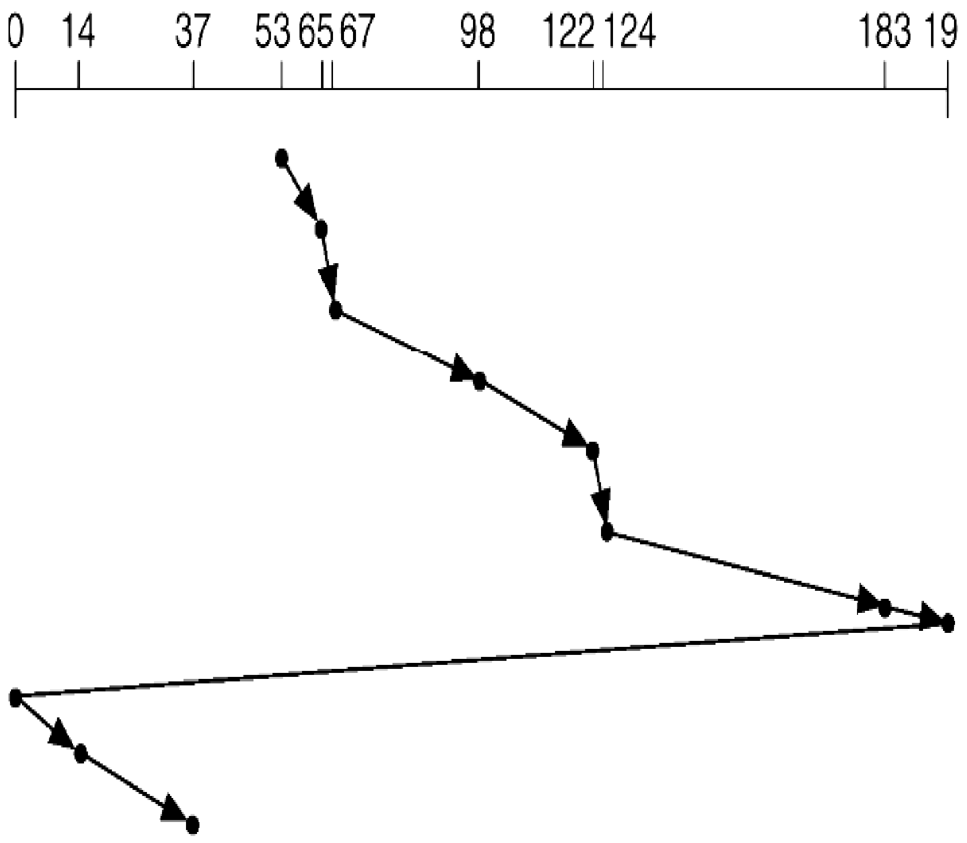


Figure 7.5 : C-SCAN disk scheduling

The total head movement (or cylinders it passes through) for this algorithm is only 183 cylinders, but still this isn't the most sufficient.

C LOOK scheduling :

Both SCAN and C SCAN algorithm moves to the end of the disk while servicing the requests. But, why do the disk head needs to go to end of the disk? Why not the disk head moves till the last request for the cylinder in each direction? In LOOK algorithm, the disk head only goes till the last request in each direction and then it reverses the head movement direction immediately without going to the end of the disk. Versions of SCAN and C SCAN which are based on this pattern of servicing requests are known as C SCAN and C LOOK scheduling respectively. Practically, the SCAN and C SCAN are often implemented as C SCAN and C LOOK algorithm.

Queue : 98, 183, 37, 122, 14, 124, 65, 67

Head starts at 53

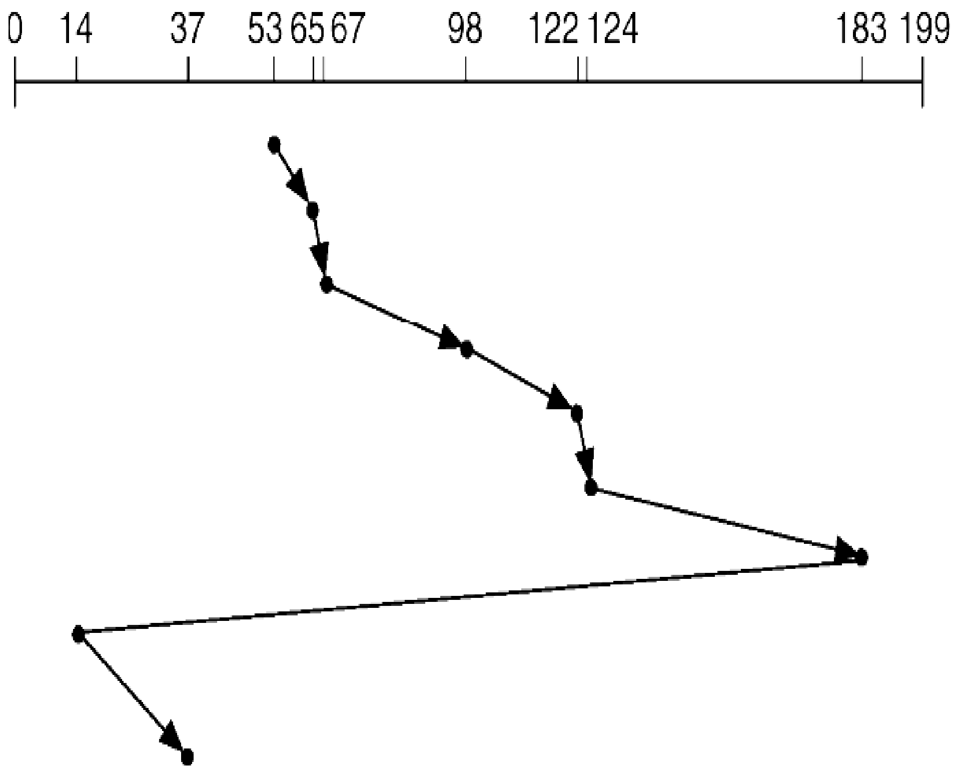


Figure 7.6 - C-LOOK disk scheduling

C LOOK prevents the extra delay which occurred at unnecessary traversal to the end of the disk having no request to further process as shown in figure 7.6. Hence C-SCAN had a total movement of 183 cylinders but C-LOOK reduced it down to 153 cylinders.

Check your progress

1. Disk requests come to a disk driver for cylinders in the order 10, 22, 20, 2, 40, 6, and 38 at a time when the disk drive is reading from cylinder 20. The seek time is 6 ms/cylinder. What would be the total seek time, if the disk arm scheduling algorithms is first-come-first-served.

2. Consider a disk queue with requests for I/O to blocks on cylinders 47, 38, 121, 191, 87, 11, 92, 10. The C-LOOK scheduling algorithm is used. The head is initially at cylinder number 63, moving towards larger cylinder numbers on its servicing pass. The cylinders are numbered from 0 to 199. What is the total head movement (in number of cylinders) incurred while servicing these requests?

7.6 FREE SPACE MANAGEMENT

Since disk space is limited, it is necessary to reuse the space freed from deleted files. Operating system maintains a free space lists to keep track of free disk blocks. These free blocks are not allocated to any file or directory. When a new file is created, the required amount of blocks is searched from the free-space list and allocated to the new file. These blocks are then removed from the free-space list. When a file is deleted, its disk blocks are added to the free-space list. Now, we will discuss the various implementation of the free-space list.

Bit-Vector

This implementation consists of an array of disk blocks entries each of 1 bit. The bit is set to 1 if a block is free or 0 if allocated. For example, consider a disk where blocks 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17, 18, 25, 26, and 27 are free and the rest of the blocks are allocated. The free-space bit vector would be 001111001111110001100000011100000 ... as shown in figure 7.7.

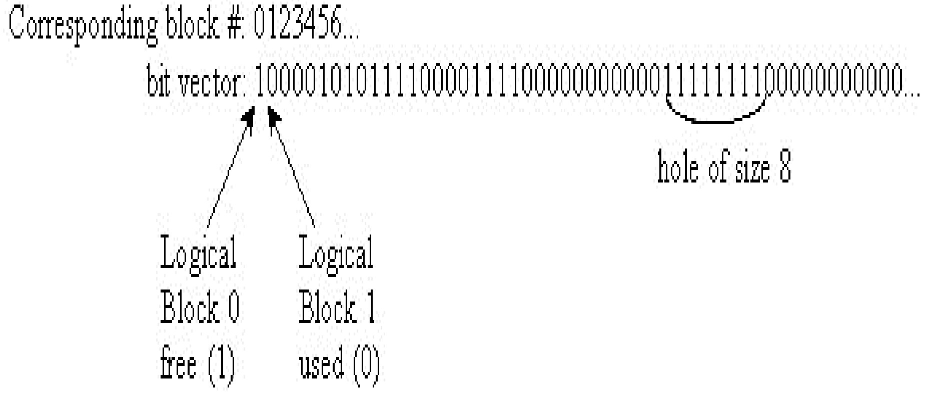

Corresponding block #: 0123456...
bit vector: 1000010101111000011110000000000001111111100000000000...
hole of size 8
Logical Block 0 free (1)
Logical Block 1 used (0)

Figure 7.7 : Bit - Vector free-space list on disk. [1]

Bit-vector is useful when it can be kept in main memory but as the disk sizes get larger, it takes more space in main memory. So Bit vector requires extra space to track free blocks of the disk. For example, a 1-GB (230 bytes) disk with 4-KB (212 bytes) blocks requires 32 KB (230/212 = 218 bits) to track its free blocks.

Linked-List

Another approach to free-space management is to link together all the free disk blocks as a linked-list. Each block contains a pointer to the next free block. A pointer to the first free block is kept in a special location on the disk and caching it in memory when needed. Considering our earlier example, in which blocks 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17, 18, 25, 26, and 27 are free and the rest of the blocks are allocated. In this approach, we keep a pointer to block 2 as the first free block. Block 2 would contain a pointer to block 3, which would point to block 4 and so on (Figure 7.8).

Problem arises when multiple free blocks are needed. Then we should follow multiple links all over the disk which results in poor performance.

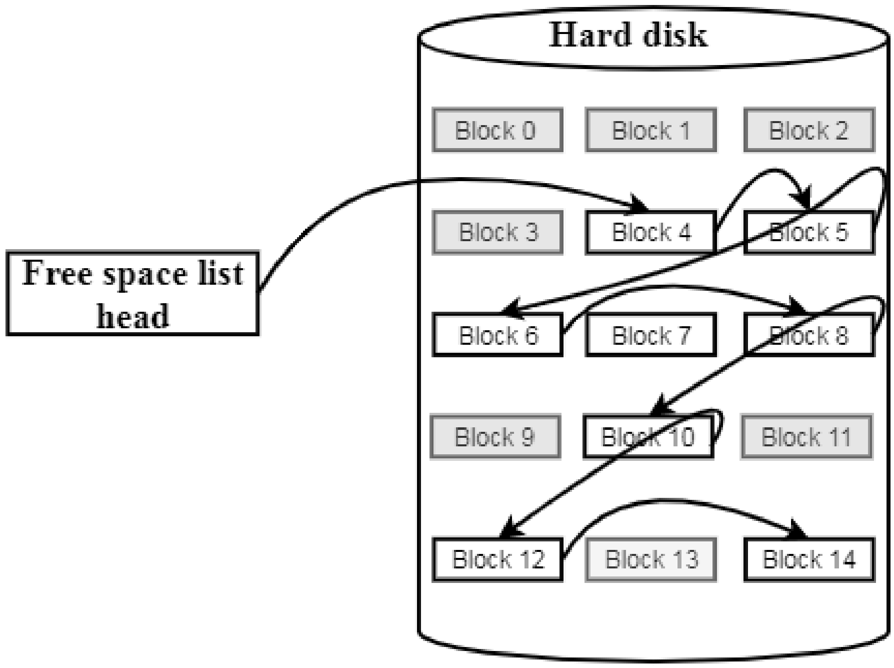


Figure 7.8: Linked free-space list on disk [1].

Grouped Linked List

Grouping is a modification of the linked-list approach. A single free block hold the addresses of a group of free blocks. The last entry in a group points to a free block which contains addresses to another group of free blocks. For example, the following free blocks-2, 5, 13, 14, 15, 23, 24, 29, 31, 37, 38, 41, ... whose grouped Linked List implementation shown in figure 7.9

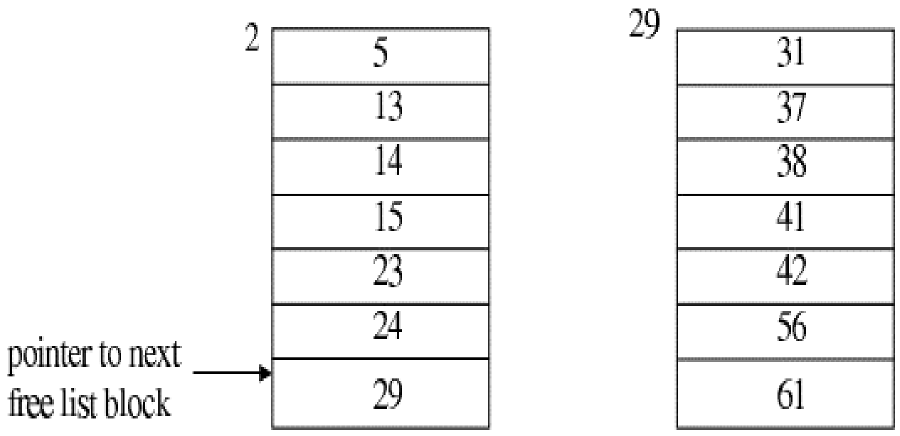


Figure 7.9 : Grouped Linked List. [1]

This approach require no disk space for implementation. It only need to store location of first pointer block.

Counting

In this approach the free space list keeps the addresses of the first free blocks and the number of free contiguous blocks that follow the first block. So each entry in the free-space list then consists of a disk address and a count. This approach is useful where, several contiguous blocks may be allocated or freed simultaneously.

7.7 SWAP-SPACE MANAGEMENT

In virtual memory management, you have learnt about swapping concept where processes are moved between main memory and disk when amount of main memory space gets critically low. In modem operating system, instead of swapping entire processes, OS swaps unused pages of processes. Virtual memory utilizes disk space as an extension to main memory and disk access time is significantly slower than main memory access time. So, use of swap space decreases system performance. In this section, we will discuss how swap space is used, where on the disk it is located, and how it is managed.

Swap space use : Different operating systems uses swap space in different ways that depend on how they implements their virtual memory. For example, swapping system uses swap space to hold entire process while paging system uses swap space to hold individual pages of processes. The swap space generally varies from few megabytes to several gigabytes. This depends on physical memory size, amount of virtual memory it supports and how virtual memory is implemented. It always good to overestimates the required swap space size for safer side. Because, if a system runs out of swap space, it may abort the process or crash the system. Earlier Linux recommended swap space size double to the physical memory size. But now, most Linux system uses considerably low swap space size. Some operating system like Linux uses multiple swap spaces. These swap spaces usually reside on separate disks.

Swap space location : A swap space of an OS may resides at any one of the two location: inside a normal file system as a file and a separate disk partition. When the swap space resides inside the normal file system as a file, normal file system routines can be used to create it, name it and allocate its space. Although, this approach is easy to implement, but it is inefficient. Because during swapping, it requires navigating directory structure and then disk allocation data structure. Also, external fragmentation may increase swapping time as it requires multiple search during swapping a process.

When the swap space resides in a separate raw partition, no file system or directory structure is placed in that partition. Instead, a separate swap space manager is used to allocate and deallocate disk blocks to a process during swapping. The swap space manager is optimized for speed rather than storage. Because, swap space is accessed more frequently than a normal file system. Although it may result in internal fragmentation but, life of data on the swap space is shorter. Also the swap space is reinitiated at every boot of the system. Some operating systems like Linux allow the swap space to reside at both raw partition and normal file system.

Swap space Management : An Example: Earlier UNIX system implements swapping in swap space management which swap-out and swap-in an entire process from physical memory and disk. Later when UNIX system evolved and paging hardwares became available, it started using combination of both swapping and paging. For example, Solaris 1 stores text segment pages in the file system. While the pages associated with stack, heap and uninitialized data of a process are stored in swap space. During an execution of a process, text segments pages are brought into physical memory from the file system. Later, when these pages are thrown away from the memory, it is more efficient to reread the pages from the file system compared to first write them to swap space and reread from there. The pages associated with stack, heap and initialized data would be written to swap space when these pages are flushed out of the memory during execution. In later version of Solaris, swap space is only allocated to a page when it is to be thrown away from memory. So, swap space is not allocated when the virtual memory page is first created. This approach gives better performance on modern computers having larger physical memory since they tend to page out less.

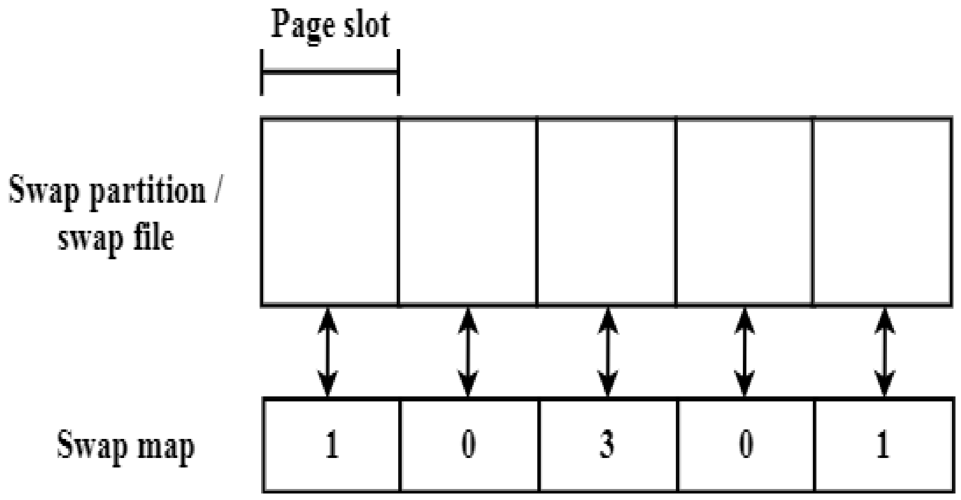


Figure 7.10 : The data structures for swap area on Linux system [1].

Linux system is similar to Solaris that allows swap space only to stores pages associated with stack, heap and uninitialized data of a process. Linux maintains one or more swap areas that reside either in a raw-swap-space partition or swap file on a regular file system. Each swap area is divided into 4-KB page-slots that holds swapped pages as shown in figure 7.10. Each swap area is associated with a swap map which is an array of integer counters. The swap map is used to map to its corresponding page slot in the swap area. A counter value of 0 indicates availability of the page slot. A counter value greater than 0 indicates that the page slot is currently occupied by a swapped page. The counter value tells how many processes are mapped to a swapped page. For example, a counter value 3 indicates that a swapped page is mapped to three different processes. In other words we can say the page is shared by three processes.

7.8 SUMMARY

In this unit we discussed

- A logical structure of a magnetic disk that consists of multiple double sided platters coated with magnetic recording material. Each platter is divided into a number of tracks and each track is further divided into multiple sectors. Data is stored on these sectors in form of blocks.
- Before a disk can store data, disk must be low- level-formatted to create the sectors on the raw hardware. Then, the disk is partitioned and an initial file-system data structures are stored on the disk. These data structures maps free space, allocated space and the root directory.
- The queue of requests for I/O blocks are serviced by a disk scheduling algorithm. The disk scheduling algorithms such as SSTF, SCAN, C-SCAN, LOOK, and C-LOOK service theses requests in to minimize the mechanical seek time of the disk and improve disk performance.
- OS maintains a free space lists to keep track of free disk space. Virtual memory uses swap
- Space located on the disk as an extension of main memory which resides either on a raw disk partition or a file within the file system.

7.9 TERMINAL QUESTIONS

1. What is the major drawback with fcfs disk scheduling algorithm?
2. Does the fcfs causes starvation of some requests?
3. Suppose a disk has 201 cylinders, numbered from 0 to 200. At some time the disk arm is at cylinder 100, and there is a queue of disk access requests for cylinders 30, 85, 90, 100, 105, 110, 135 and 145. If Shortest-Seek Time First (SSTF) is being used for scheduling the disk access, the request for cylinder 90 is serviced after servicing ____________ number of requests.
4. What is the disadvantage of the SSTF disk scheduling algorithm?
5. How does starvation of requests possible in SSTF disk scheduling algorithm.

6. Discuss the disadvantage of SCAN disk scheduling algorithm.
7. How the SCAN does differs from C SCAN disk scheduling algorithm.
8. How the LOOK algorithm does handles the problem with C SCAN algorithm.
9. What are the purposes of swap space in OS?

UNIT-VIII CASE STUDY OF UNIX

Structure

8.1 INTRODUCTION

UNIX has a very special place in the history of operating system. The first implementation of UNIX was developed at Bell Telephone Laboratories in early 1970 by Ken Thompson and Dennis Ritchie. It was first used inside the BELL lab and soon it was licenced to use in universities and various research purpose at low cost. UNIX was written in c language that was the first widely used operating system portable to various computer architecture. So, during 1970s, UNIX became most common operating system used in universities by students and faculty for class work, project and operating system research. Since, UNIX source code was made available for modification, many improvements and variations of UNIX came that time. The most well-known variation of UNIX was developed at Berkeley Software Distribution (BSD), which provide many technical improvements in UNIX, including a faster file system, virtual memory, TCP/IP support, and more. AT & T (owner of BELL labs) also developed a version of UNIX called System V and its latest version is release 4 which is usually known as SVR4. Many commercial vendors have developed their own version of UNIX (for example, SUN Solaris, HP UNIX, Digital UNIX, IBM AIX) but these are either derived from BSD version or System V version. For a number of years, UNIX is the basis for operating system research and also for the technical developments in operating systems originated from UNIX. UNIX system call interface and implementation is so influential that most of operating systems developed in last 25 years have borrowed to some degree from UNIX.

8.2 OBJECTIVE

After going through this unit you should be able to

- Explore the history of the UNIX operating system and the designing principle of Linux.
- Explain scheduling of processes and inter-process communication in UNIX.
- Describe how does memory management is performed in UNIX.
- Understand how file systems are implemented in UNIX.

8.3 UNIX FEATURES

UNIX operating system supports following features.

1. Multi-user system - UNIX allows multiple users to use a same computer. A single powerful UNIX server is connected to different terminals, Keyboards, Monitors etc. and each user works with his/her own terminal.
2. Multi-tasking system –UNIX support Multitasking that allow you to perform various task at the same time by switching among these task so fast that cannot be noticed any delay.
3. Programming Facility—UNIX provides you a UNIX shell that contains all necessary programming elements like operators, many built in functions, conditional and control structures etc. to deliver you a better programming environment.
4. Security—UNIX provides read, write, execute etc. permission to each file. It encrypts the files into unreadable format. UNIX offers a login name and password to every user to protect each user files. It is impossible for a user to access another user's data.
5. Portability—UNIX is portable to different hardware platforms or hardware architecture.

8.4 STRUCTURE OF UNIX OS

UNIX OS functionalities are organized into three levels :

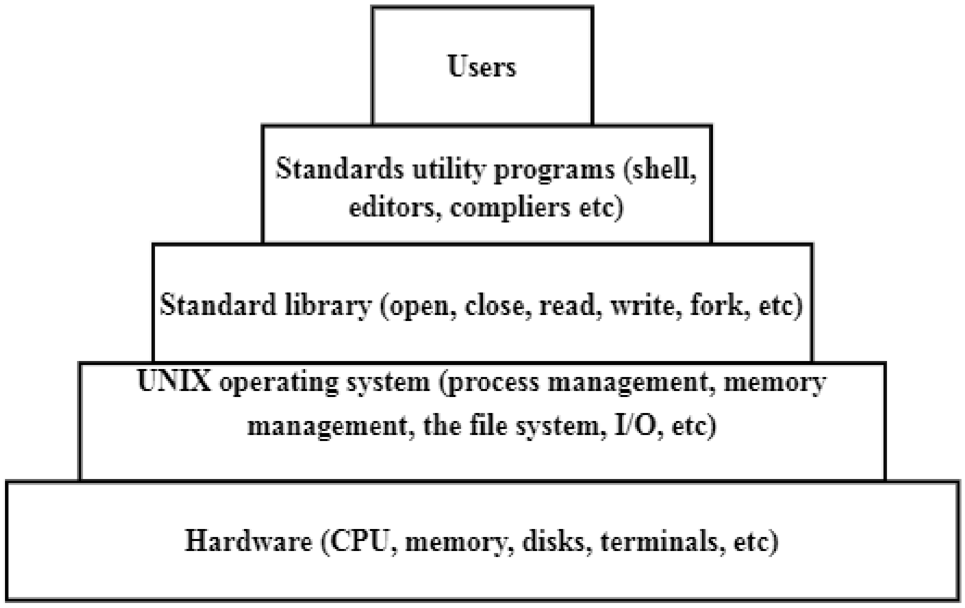

Users
Standards utility programs (shell, editors, compliers etc)
Standard library (open, close, read, write, fork, etc)
UNIX operating system (process management, memory management, the file system, I/O, etc)
Hardware (CPU, memory, disks, terminals, etc)

Figure 8.1 : Different Layers in a UNIX system.

Kernel : It is the part of operating system that implements most basic functions. All kernels are operating system but not all operating systems are kernels. UNIX kernel directly interacts with the hardware. The kernel provides the desired service as requested by user programs. User programs interact with the kernel through system calls. These system calls request the kernel for various services including creating, suspending, or killing a process; opening, closing, reading, writing or executing a file; changing ownership of a file or directory; accessing hardware devices.

Shell : The shell provides you an interface to the kernel. Shell communicates with a user through terminal and it communicates with kernel through system call. UNIX OS offers you multiple shells including the Bourne shell (sh), the Bourne Again shell (bash), the Korn shell (ksh) and the C shell (csh). Each of the shells has different shell commands, but OS commands are same in all the shells. You can type commands directly into the terminal, or can create a text file that contains a series of commands to be executed by the shell. When a series of commands are written in a text file, it is called a shell script. These commands are also a type of programs. A command typed in shell is forwarded to the kernel. Once the command is executed, control is returned to the shell. The shell then displays another prompt ($ on our systems) waiting for another command to be entered. When you type a command in the shell, it extracts the first word and assumes it to be the program name to be run. It searches for this program and run it, if it finds the program. The shell suspends itself until the program terminates. After that, the shell is back to running state and it tries to read next command. This way, shell is an ordinary user program that has ability to execute other programs by reading commands from terminal and writing results to the terminal. Commands may contain arguments that are passed to the called program. For example, following command invokes the cp program with two arguments: src and dest.

```
cp src dest
```

The program takessrc as the first argument which is a name of an existing file and copies it to dest folder.

System Utilities and Application: System utilities are specialized software (commands) which are designed for a specific task. The ls command is used to see a list of the files present in a directory. The cat command is used to ‘display content of several files to a standard output, one after another. You can use rm command to removed files. Access permissions of files can be changed by the file‘s owner through chmod command. The mkdir and rmdir command are used to create and remove directories respectively. Table 8.1 shows some UNIX utility programs along with a short description of each one.

Program	Typical use
cp	Copy one or more files
cut	Cut the columns of a text files
Make	Compile a file to build a binary
paste	Paste columns of text into a file
pr	
sort	Sort the content of a file line by line alphabetically
head	Extract the first line of a text file
tail	Extract the last line of a file
tr	Translate between a character sets
pr	Format a file for printing
grep	Search a file for some pattern

Table 8.1 : Few of the common UNIX utility programs

Some commonly used application programs include viand emacs (text editors), gcc (a C compiler), g++ (a C++compiler), latex (a powerful typesetting language), xfig (a drawing package).

8.5 PROCESS MANAGEMENT

i. **Scheduling :** System V, Release 4 of UNIX is considered to be standard UNIX. It uses a priority scheduler with 160 priority levels. Processes at levels 0-59 are interactive processes that belongto time-sharing classes. Processes at level 60-99 are for system priorities which run in kernel mode. Processes at level 100-159 belongs to real time classes. Processor always runs the highest priority process, while processes with same priorities are scheduled in round-robin fashion. The priority of a time sharing process goes down, if it uses up its time quantum and its priority goes up, if it waits for an I/O event. This time quantum varies from 100 ms (for priority 0) to 10 ms (for priority 59) as shown in Figure 8.2.

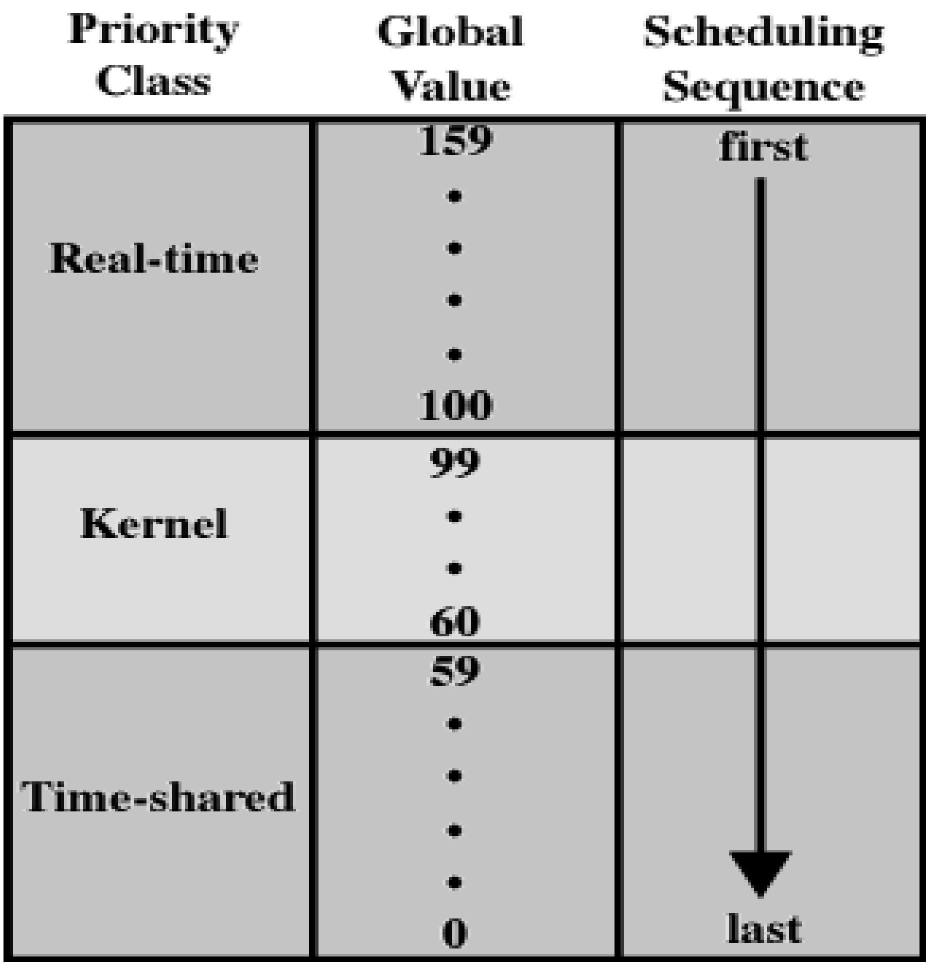


Figure 8.2 - SVR4 Priority Classes[4]

ii. **Inter-process Communication and Synchronization :** In UNIX, one process notifies another process about an occurrence of an event through signal. A signal is an event notification that one process send to another process. UNIX defines approx. 30 different events that can be signalled. Some signals are sent to a process when it gets an interrupt such as arithmetic error, instruction error and addressing error. Some signals are external events such as interrupt (when user type ctrl+c on keyboard), kill(one process destroy another process), child (status of a child process has changed), alarm(timer set by a process has expired) and I/O (I/O possible on open file). Some signals are for other errors like syscall (invalid argument to system call). When an event occurs, the currently running process is stopped and control is passed to a signal handler. The signal handler tells the system to call the procedure when a signal arrives.

Signal	Number	Description
1	SIGHUP	If a process is being run from terminal and that terminal suddenly goes away then the process receives this signal.
2	SIGINT	It is generated by the terminal when we press the interrupt key (Control-C) on the keyboard. This signal is often used to terminate a process that produces lot of unwanted output on the screen.
3	SIGQUIT	Quit from keyboard
4	SIGILL	This signal indicates that a process has executed an illegal hardware instruction that the CPU cannot understand.
5	SIGTRAP	This signal is used mainly within the debuggers and program tracers.
6	SIGABRT	The program called the abort 0 function. This is an emergency stop.

6	SIGIOT	Input/output transfer
7	SIGBUS	Bus error. An attempt was made to access memory incorrectly. This can be caused by alignment errors in memory access etc.
8	SIGFPE	Floating-point error.
9	SIGKILL	Kill signal from system. The process was explicitly killed by somebody wielding the kill program.
10	SIGUSR1	User defined. It is upto the programmer what they want to do.
11	SIGSEGV	An attempt was made to access a memory that is not allocated to the process.
12	SIGUSR2	Left for the programmers to do whatever they want.
13	SIGPIPE	Pipe fault (broken pipe). This signal is sent to inform a producer process when the consumer process died and not available to consume the output that would be fed via a pipe.
14	SIGALRM	A process can request a "wake up call" from the operating system at some time in the future by calling the alarm( ) function. When that time comes round the wakeup call consists of this signal.
15	SIGTERM	The SIGTERM signal terminates a process. Unlike SIGKILL, this signal can be blocked, handled, and ignored. It is the normal way to

		politely ask a program to terminate.
16	SIGSTKFLT	Stack fault
17	SIGCHLD	When a child process terminates or stops, SIGCHLD signal is sent to the parent process to know the status of the chid process.
18	SIGCONT	Sending SIGCONT to a process, resume its execution that is previously paused by SIGSTOP.
19	SIGSTOP	It is used to pause a process.

Table 8.2 UNIX Signals

Some of the UNIX signals are listed in Table 8.2. Signals in UNIX allow one process to signal an asynchronous event to another process, while pipes allow one process to send data to other process. SVR4, also implements message queues and message sending where message can be of any length. SVR4 implements shared memory which allows two processes to set up a part of their address spaces to be shared between two processes. Shared memory is a fast way of transferring a large quantity of data. Solaris provides “system V semaphores” to control access to shared resources. A semaphore is used to lock shared resources prior to its use. It releases the resources when a process completes the use of the resources.

iii. **Booting in Linux :** OS loads programs into the main memory for its execution. But, how does OS get loaded into the memory. The process of loading OS into memory is given as follow. When you turn on your computer, a small program stored on ROM called BIOS is executed which perform POST (power on self-test) i.e. devices discovery and their initialization. This initial devices check-up is required because the loading of OS depends on these devices. This BIOS program performs reading and writing operation. The BIOS loads the first sector of the boot disk called MBR (master boot record) into the memory and transfers its control to it. The MBR contains bootstrap program or bootloader (GRUB). This program searches OS kernel into the root directory of the boot disk. The boot program loads the OS kernel into the memory and gives control to it. This whole process is called bootstrapping or booting.

Check your progress

1. What are the important features of UNIX operating system?
2. What do you mean by kernel in UNIX system?
3. How the kernel is differ from shell and system utilities?
4. How does UNIX system perform interprocess communication and Synchronization?

8.6 MEMORY MANAGEMENT

Memory management is responsible for allocating a free portion of memory for a new process, deallocating parts of memory when they are freed, keeping track of parts of memory that are in use, demand paging where main memory is not sufficient to hold all the processes and process swapping between main memory and disk. Many memory management schemes were proposed, but the swapping and paging were two of the most important ones that still significant to modern operating systems. The swapping mechanism was first adopted in AT&T Bell Lab (Ritchie et al 1974) version of UNIX System. The paging technique was added to UNIX BSD variants (University of California at Berkeley). In this section, we will discuss in detail about the UNIX memory management schemes including process swapping and demand paging.

Physical vs. Virtual Memory :

Swap space is designed where physical memory is insufficient for the current process. UNIX allows you to use some portion of disk space along with the physical memory installed in your system along with as a swap space. The Virtual memory is the sum of the physical memory (RAM) and the total swap space assigned by the system administrator at the time of system installation.

Virtual Memory (VM) = Physical RAM + Swap space

The size of swap space is usually two or three times of the main memory that is solely reserved for swapping. A new process cannot start if there is no available swap space for it. That is why sometime virtual memory system gives “out of memory” message which means the system is currently out of swap space. Thus, a complete image of the memory of every process is kept in swap area.

Demand Paging : UNIX divides physical memory into equal sized blocks called frames. Each process is also divided into blocks of the same size called pages. The virtual memory address space consists of logical page number and displacement within the page. The virtual page number is translated into real page frame and the displacement indicates the byte offset into that page. When a page of a process is put in a frame, this page is really allocated in physical memory. UNIX kernel uses four major data structures to support demand paging: page table, frame table, and swap table, Disk Block Descriptor.

Page Table : Each entry of a page table is indexed by a page number of the process. The page table consists of many entries and each entry has several fields as shown in Figure 8.3

Page frame number	Age	Copy on write	Mod-ify	Refe-rence	Valid	Pro-tect

Figure 8.3 : Fields of entry in the page table

Page frame number : It is the physical address of a page in the main memory.

Age : These bits indicate how long the page has been in memory without being referenced. This information is used for page replacement.

Copy on write : This bit is set when a page is shared by more than one process. When more than one process shares a page and one of the process tries to write into the page, a new copy of the page must be made first for all other processes that share the page.

Modify : This bit tells whether the page has been recently modified or not by the processes.

Reference : This bit indicates whether a page has been referenced or not, either for reading or writing. When a page is first loaded into the memory, this bit is set to 0. The bit periodically changes as it is referenced. This information is used by the page replacement algorithm.

Valid : It shows a page is present in the main memory or not.

Protect : These bits indicate what kinds of access (read, write or execute) is permitted to the page. For example, if it consists of 3 bits, each of the one bit indicates access permission for reading, writing or executing the page.

Frame Table

Frame table is used to free some frames from physical memory to make it available for other process. Each frame in the physical memory has an entry in the frame table. So, the table is indexed by frame number as shown in Figure 8.4.

Page state	Reference count	Logical device	Block number	Pfdata pointer

Figure 8.4 : Fields of entry in the frame table

- **Page state :** It tells whether this frame is available for reallocation or its associated page is stored on swap device. If the frame has an associated page, the status of the page is specified i.e. whether it is on swap device or executable file, or DMA in progress.
- **Reference count :** it is the total number of processes that refer the page.
- **Logical device :** It references the logical device where the page is stored.
- **Block number :** It contains block number where the page is stored on the logical device.
- **Pfdata pointer :** Pointer to other pfdata table entries on a list of free pages and on a hash queue of pages.

Swap Table

The swap table contains an entry for each page on the Swap Device. The swap table (shown in Figure 8.5) is used by page replacement and swapping.

Reference count	Page/storage unit number

Figure 8.5 : Fields of entry in the frame table

- **Reference count :** This refers to number of page table entries that corresponds to a same page on a swap device.
- Page/storage unit number: Page identifier on storage unit.
- **Disk Block Descriptor :** Each page of a process is associated with an entry in Disk Block Descriptor that tells location of disk copy of the page. Processes that share common pages have same entries in their Disk Block Descriptors.

Swap device number	Device block number	Type of storage

- Swap device number: It is a Logical device number of the secondary device which holds the page. More than one device to be used for swapping.
- Device block number: Location of the block containing the page on swap device.
- Type of storage: This may be a swap device or executable file.

Page Replacement :

In virtual memory management, the demand paging cooperates with page replacement. To keep executing a progress, its pages in memory are replaced by its new pages. So, the working pages of a process change dynamically. The Frame table is used for page replacement when no frame is available for new pages of a process. The kernel maintains a list of free frames available to bring these pages. The kernel takes out frames from this free frame list, when number of frames in physical memory drops certain threshold. SVR4 version of UNIX uses a two-handed clock algorithm as shown in Figure 8.6. The algorithm works by using the reference bit of the page table entries whose corresponding pages are in memory. When a page is first time loaded into the memory, its reference bit is set to 0. Later, it is set to 1 if it is referred for either reading or writing. The fronthand of the clock algorithm, sweeps through a list of eligible pages for swapping out and set their corresponding reference bit to 0. After Sometime, the backhand scans through the same list of eligible pages and checks their reference bits. If their reference bits are set to 1, these pages have been referenced recently since they are swept by the fronthand. So, these pages are then ignored. If their reference bits are still set to 0, then they are placed on a list to be paged out.

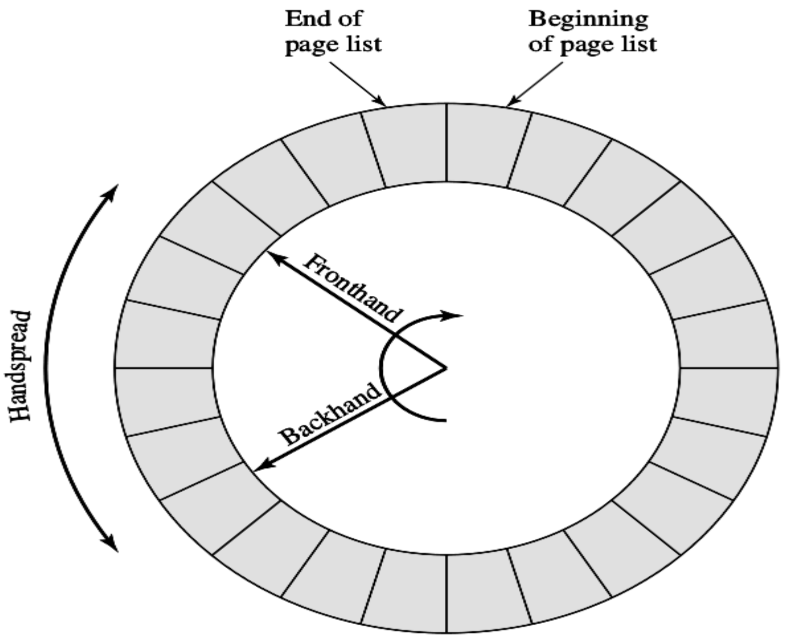


Figure 8.6 : Two-Handed Clock Page Replacement Algorithm[4]

The algorithm has two parameters to controls its operation:

- **Scanrate :** It is the rate at which the two hands scan through the list of eligible pages. It is measured in pages per second
- **Handspread :** The gap between fronthand and backhand of the clock algorithm.

These parameters have default values at boot time based on the availability of physical memory. The scanrate varies linearly between slow scan and fast scan as the amount of free memory vary. The clock hands move more rapidly to free up more pages, if the free memory shrinks. The handspread along with scan rate give opportunity to a page to be used again before it is swapped out due to its lack of use.

Check your progress

1. Briefly explain how UNIX system performs memory management.
2. Explain various data structures used during demand paging in UNIX system.

8.7 FILE SYSTEM

A file system is an abstraction that provides support for creation, deletion, modification of files, and organization of files into directories. It also provides support for access control to files and directories and efficiently manages the disk space. A UNIX file system is a data structure resident on disk. It contains a super block that defines the file system, an array of inodes that define the files in the file system, the actual file data blocks, and a collection of free blocks. All allocation is performed in fixed-size blocks. File names are appeared in directories. Each directory contains names of its files and their pointer to corresponding Inode. An Inode of a file contains pointer to all disk blocks of the file. The detailed structure of the Inode will be discussed later in this section. UNIX systems use a hierarchical file system with the root node as its origin. This hierarchical tree like structure has a single parent directory and its sub ordinate directories can have many child directories as illustrated Fig. 8.7.

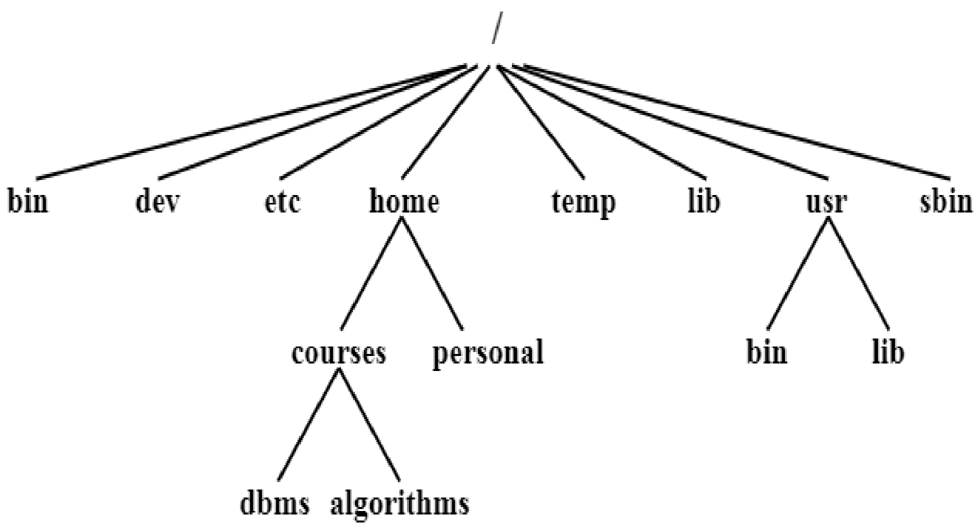


Figure 8.7: A typical UNIX filesystem tree.

Directory	Typical Contents
/	The "root" of the file system.
/bin	This directory contains system binary files for most commonly used executable commands like cp (file copy), rm (file delete), ls (list files in directory), etc. generally needed by all users.
/usr/bin	It contains higher-level system utilities and application programs for users
/sbin	This directory contains system utilities for performing system administration tasks by user administration.
/lib	Contains system libraries files such as kernel modules or device drivers.
/usr/lib	It stores the required libraries and data files for programs stored within /usr or elsewhere.
/tmp	It is a place for temporary files that are periodically removed from the file system. For example, many systems clear this directory upon startup.
/home or /homes	It is user home directories containing personal file space for each user. Each directory is named after the login of the user.

/etc	Contains UNIX system configuration files and databases.
/dev	Contains device specific files for every peripheral device attached to the system.
/proc	A pseudo-filesystem which is used as an interface to the kernel. Includes a sub-directory for each active program (or process).

Table 8.3 : Some common directories on UNIX systems.

File names appear in directories and a pointer to the file's inode that contains pointers to all disk blocks of the file. The detailed structure of an inode will be discussed in the next section. In UNIX, there are three kinds of files: ordinary disk files, directories, and special files.

i. **Ordinary files :** Ordinary files can be symbolic or binary files. You can place any information in a file and the system is not expecting any particular structuring. A text file consists a sequence of characters with new lines. Binary programs or files contain non-numeric text which is mostly numeric data as sequences of word. These sequences of words appear in main memory, when the program starts executing. A few user programs manipulate files and control its structure. For example, the assembler generates, and the loader expects, an object file in a particular format. A file name is suffixed with an extension to indicate the nature of the file's contents (such as .jpeg for image, .txt for text file, .p for Pascal code and .c for C source code). Extensions are the set of conventions established by users and programs whose use is not required by the operating system. It provides an easy way for users to see type of a file at a glance and group similar files together so that they can be copied, moved or deleted as a group.

ii. **Directories :** Directories provide the mapping between the names of files and the files themselves. A directory is similar to an ordinary file and implemented in same way except a restriction that it cannot be written on by unprivileged programs. However, users can read from directories. There is a login directory for each user and the user may also create sub-directories to group some files together. The system maintains several directories for its own use. One of these is a root directory. All files in the system can be found by tracing a path through a chain of directories with the "root" directory as its origin. Other system directories contain all the programs or commands for general use.

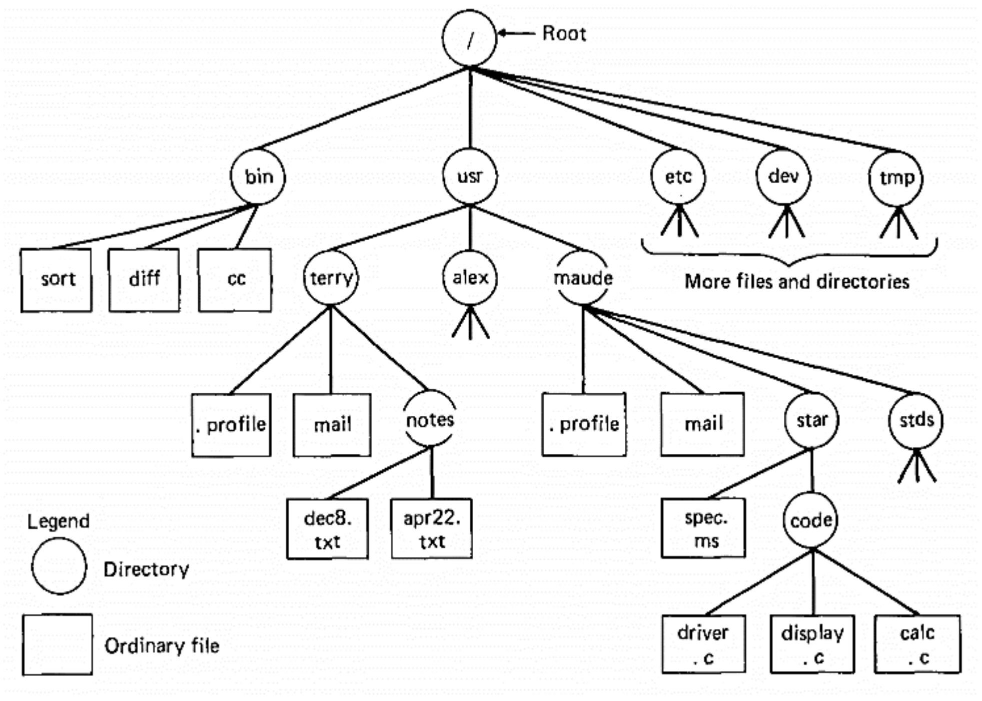


Figure - 8.8 UNIX file system hierarchy[6]

A file name is specified to the system in a form of a path name, which is a sequence of directory names each one is separated by slash, ‘‘/’’, and ends with the file name. If the sequence starts with a slash, the file search starts from the root directory. For example, a name /usr/terry/notes/apr22.txt (see figure 8.8) causes the system to start search the root for directory usr, then to search usr for terry, and then to search terry for notes and finally to search notes for apr22.txt. The name ‘‘/’’ refers to the root itself. A path name not starting with ‘‘/’’ causes the system to start a search with current working directory of user. At any time, the system knows user’s current directory or working directory. Thus, a path name notes/apr22.txt specifies the file apr22.txt in a subdirectory notes that appears in the current directory. A file name itself, for example, apr22.txt, refers to a file that can be found in the current directory. A null file name refers to the current directory. Directory entries for files are called links. The links in directory entries point to inodes which, in turn, point to the actual physical files. A file will disappear when the last link to it is deleted. Each directory always has at least two entries i.e‘‘ . ’’ and ‘‘ .. ’’ . The name ‘‘ . ’’ in each directory refers to the directory itself. The name ‘‘ . . ’’ refers to the parent to a directory in which it appears or it was created.

iii. **Special files :** There exist special files for each disk, tape drive, magnetic tape, physical memory etc. These special files usually reside in /dev directory and they are associated with each system supported device. Read and write to these special files is done similar to the ordinary disk files, but these actions also cause activation of the associated devices. For example, to write on a magnetic tape drive, the kernel writes on the file located at

/dev/mt. Special files exist for each disk, each tape drive, and for physical main memory. The file directory structure shown above appears as a single file system, but it may be viewed as many separate file systems under the root directory and subordinate to it. These file systems may reside on either the same disk as the root of the file system or on separate disks, but any file system must reside on only one disk.

Removable file systems : UNIX systems have a single directory tree with the root of the file system is always stored on the same device. So it is not necessary that the entire file system hierarchy reside on this device. All accessible storage must be associated to this single directory tree, before they are used. This is unlike Windows where there is one directory tree for each storage component (drive). If you have a directory in a file system, you can mount another file system into that directory. Mounting is an act where a storage device is associated to a particular location in the directory tree. For example, during the system initialization, the user file sytem on the device /dev/userdisk is mounted on /user directory of the root file system as shown in figure 8.9.

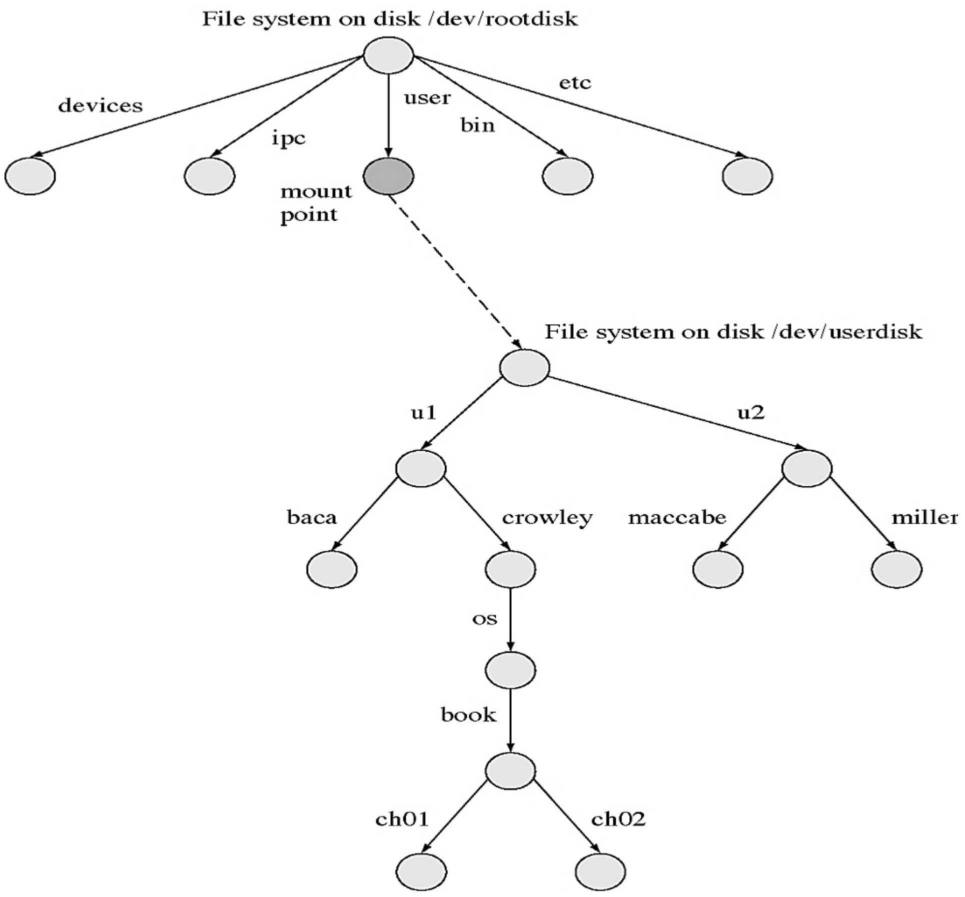


Figure-8.9- Typical structure of UNIX file system.

Mounting a file system in UNIX requires calling a mount system call. The mount system call takes two arguments: name of the existing ordinary file, and name of the special file associated to a storage volume (e.g., a disk pack). The special file may have independent file system which may contains its own directory hierarchy. After mount, reference to the file system on removable volume requires referring to the ordinary file. Mount replaces the leaf containing the ordinary file in the hierarchy tree by a whole new subtree (or the hierarchy) stored on the removable volume. After the mount, there is virtually no difference between the files on the removable volume and those on the permanent file system.

UNIX Inode : UNIX Inode is a data structure that describes a file or directory in UNIX system. It contains important information related to a file within a file system. The UNIX Inode contains 13 block addresses as shown in figure 8.10. The first 10 (0-9 block address) are direct block address of first 10 data blocks of a file. The 11th field contains a one level index block address which points to a block containing a group of block addresses. This block normally contains 256 block addresses (block size/block address size). The 12th field contains a two- level index block address. This block points to a block containing one level index block addresses which, in turn, points to blocks containing direct block addresses. 13th field contains a three-level index block address which points to a block containing two level index block addresses. Each entry in this block, points to a block of one level index block addresses. Each entry in these blocks point to a block of direct block addresses. Each entry in these blocks ultimately points to a block of the file.

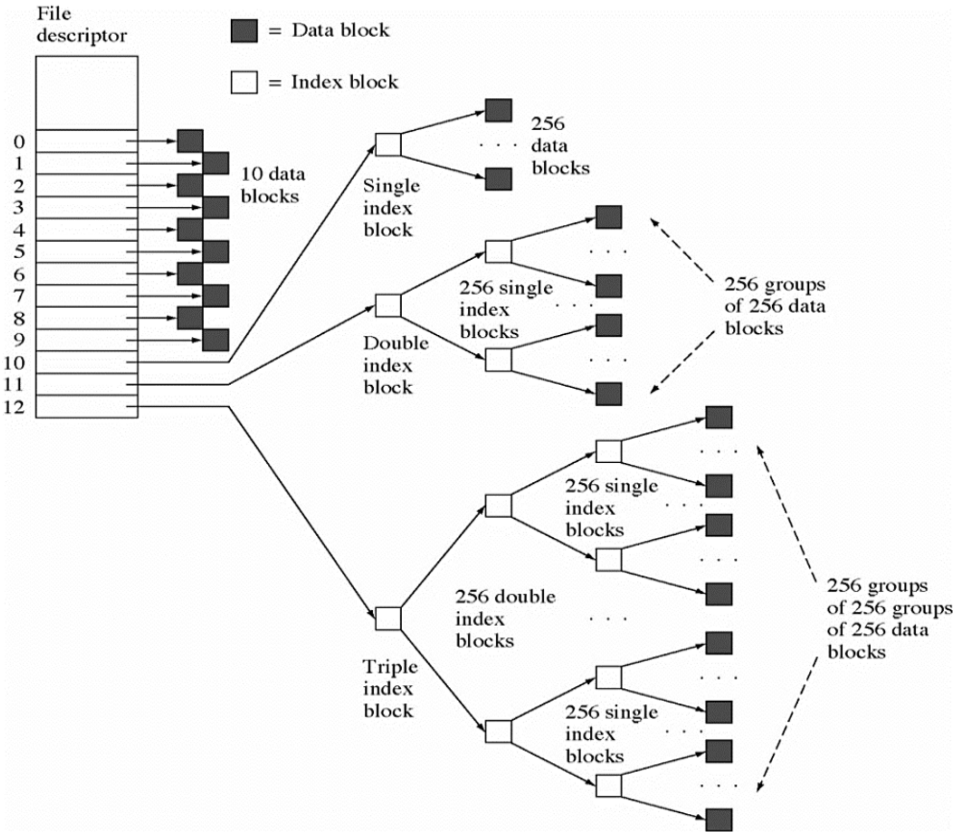


Figure 8.10 : UNIX Inode to allocate every block in a file[2]

File and Directory Permissions:

UNIX offers three types of permission to each file or a directory on a UNIX system. These permissions describe what operations could be formed on these files and directories by various categories of users. The permissions assigned to Files and directories differ slightly as both entities are different from each other. The interpretation of the permissions assigned to files and directories are shown in the Table 8.4.

Permission	File	Directory
Read	You can see the content of file	You can list the files in the directory
Write	You can modify the content of the file	You can create or remove files in the directory.
Execute	You can execute the file as a program	You can enter into the directory and access other files and directory in them.

Table 8.4 : Permissions for files and directories

A UID (User ID) is an integer that describes the owner of a file. Each files and directories are marked with the UID that is the owner of the file. User can be organised into groups called GID (group IDs). Users are assigned to different groups manually by the system administrator. Each file carries the UID and GID of its owner. When a new file is created, it gets a set of permissions by the creating process. Theses permission describes the type of accesses by the owner, other member of the owner's group and rest of the users on the files. Since there are three categories of users and three types of operations (read, write and execute) for each type of user, 9 bits required to specify access permission. So, permissions are a set of characters that describe access rights. There are 9 permission characters that describe three access types to each of three user type. The three access types are read ('r'), write ('w') and execute ('x'). The three user categories include: the user who owns the file, other users in the owner's group and rest of the users (the general public). The 'r', 'w' or 'x' character in a permission corresponds to the present of its respective access right while '-' corresponds to absence of any access right. For example, some set of a 9 permission characters and their meaning are shown in Table a file with write permissions for the owner, read and write for the group and execute for others will show as Table 8.5.

File permissions	Allowed file accesses
rwxrwxrwx	All users have read, write, and execute permissions.
r--------	The owner has read permission and no permission for other users.
rw-------	The file's owner has read and writes permissions and no permission for other users.
rwx------	The file's owner has Read, write, and execute permissions while, all others have no access.
rw-rw-rw-	All users have read and write permission while, no execute permissions for anybody.
rwxr-xr-x	All users have read and execute permission while, the file's owner also has write permission.

Table 8.5 : Some example of file permissions

The permission of a file or directory can only be modified by its owners, or by the superuser (root) using the chmod system call.

Syntax :

$ chmod options files

The options for Chmod command can be specified in two ways. Firstly, permissions may be specified as a sequence of 3 octal digits (octal number system has the digit range 0 to 7) as shown in table 8.6. Each octal digit represents the access permissions for the user/owner, group and others respectively. The mappings of permissions to corresponding octal digits are shown in Table 8.6.

Permissions	Mapping
---	0
--x	1
-w-	2
-wx	3
r--	4
r-x	5
rw-	6
rwx	7

Table 8.6 : Mappings of permissions to their octal digits

For example, the following command sets rw------- permission on file account.txt:

```
$ chmod 600 account.txt
```

Secondly, the options for the file can also be specified symbolically. The general syntax is:

```
$ chmod X@Y file
```

In above syntax, X represents any combination of the letters: ‘u’ (for owner), ‘g’ (for group), ‘o’ (for others), ‘a’ (for all; that is, for ‘ugo’). @ symbol represents either ‘+’ to add permissions, ‘-’ to remove permissions, or ‘=’ to assign permissions absolutely. The Y here represents any combination of ‘r’, ‘w’, ‘x’ access rights. For example, the table 8.7 shows some typical chmod commands and their descriptions:

Commands	Descriptions
chmod u=rx file	Grant read and executes but not writes permissions of file to the owner.
chmod go-rwx file	Remove read, write and execute permission of group, others users.
Chmod g+w file	Grant writes permission to the group.
Chmod a+x file	Grant executes permission to everybody.

Table 8.7 : Some examples of chmod commands.

8.8 SUMMARY

- In this unit, we discussed the history of the UNIX operating system and the principles upon which Linux is designed.
- We explained how UNIX schedules processes and provides interprocess communication. We have seen, SVR4 uses a priority scheduler with 160 priority levels. The process with highest priority is always running and processes with same priorities are scheduled in round-robin fashion.
- We discussed, in SVR4, one process signal an asynchronous event to another process through Signal and one process send data to other process through pipes and message queues. SVR4 implements shared memory that allows two processes to set up a part of their address spaces as shared between these two processes.
- We looked at memory management in UNIX. The Virtual memory is the sum of the physical memory (RAM) and the total swap space

assigned by the system administrator at the time of system installation. Virtual memory is commonly implemented by demand paging. Demand paging in SVR4 version of UNIX uses three data structures - page table, frame table, and swap table. The page replacement algorithm used in SVR4 is a refinement of the Least Recently Used (LRU) which is also known as the two-handed clock algorithm. In two-handed clock algorithm, the backhand follows the first hand by some distance and allows pages to be reclaimed sooner than waiting for a full revolution of the fronthand.

- We explored that, the UNIX organises the file system as a hierarchical tree like structure with a single root directory at the top and every non-leaf node of the file system structure is a directory of files. Files at the leaf nodes of the tree are directories, regular files or special device files. Every file has one inode, but it may have several names called links, all of which map into the inode. Inode list is stored in the file system on the disk and the kernel reads them into main memory when accessing files.
- UNIX provides protection and security of files and directory by controlling read, write and execute access. File and directory permissions can only be modified by their owners or by the superuser (root) using the chmod system call.

8.9 TERMINAL QUESTIONS

1. What are the purposes of system call in UNIX system?
2. Explain following terms:
   a) Inter process Communication
   b) Synchronization
   c) Two handed clock algorithm
3. Explain utility programs and application program with examples.
4. How the process scheduling takes place in UNIX system.
5. Describe the types of files in UNIX system.
6. What is the purpose of Unix Inode? Describe the structure of Unix Inode.
7. What are the various file and directory permissions in UNIX system?

BIBLIOGRAPHY

1. Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. Operating System Concepts (8th. ed.) : Wiley Publishing, 2008.

2. Crowley, Charles. Operating systems: a design-oriented approach. McGraw-Hill Professional, 1996.
3. Modern Operating Systems Second Edition by Andrew S. Tanenbaum Publisher: Prentice Hall Ptr
4. Stallings, William. Operating systems: internals and design principles. Boston: Prentice Hall, 2012.
5. Ritchie, O. M., and Ken Thompson. "The UNIX time-sharing system." The Bell System Technical Journal 57.6 (1978): 1905-1929.
6. Deitel, Harvey M., Paul J. Deitel, and David R. Choffnes. Operating systems. Pearson/Prentice Hall, 2004.

ROUGH WORK