Syllabus MCA-404 Operating System - II

Review of basic concepts of operating system, threads; inter process communications, CPU scheduling criteria, CPU scheduling algorithms, process synchronization concepts, critical section problem, synchronization hardware, semaphores, classical problems of synchronization, monitors.

Basic concepts of deadlock, deadlock prevention, avoidance and detection, recovery from deadlock, paging, segmentation, demand paging, thrashing.

File concepts, file access methods, directory structure, allocation methods, free space management, directory implementations, disk structure, disk scheduling.

Network operating systems, distributed operating systems, Introduction to UNIX operating system, Basic architecture, memory management and file handling systems.

Lesson Plan

Lesson 1: Review of basic concepts of operating system, threads; inter process communications, CPU scheduling criteria, CPU scheduling algorithms,

Lesson 2: process synchronization concepts, critical section problem, synchronization hardware, semaphores, classical problems of synchronization, monitors.

Lesson 3: Basic concepts of deadlock, deadlock prevention, avoidance and detection, recovery from deadlock,

Lesson 4: paging, segmentation, demand paging, thrashing.

Lesson 5 & Lesson 6: File concepts, file access methods, directory structure, allocation methods, free space management, directory implementations,

Lesson 7: disk structure, disk scheduling.

Lesson 8: Network operating systems, distributed operating systems,

Lesson 9: Introduction to UNIX operating system, Basic architecture, memory management and file handling systems.

Lesson Number: 1	***Writer: Dr. Rakesh Kumar***
Basics of OS & Process Scheduling	***Vetter: Dr. Pradeep Bhatia***

1.0 Objectives

The users of this lesson have already gone through some concepts of operating system in the first year of the course. So the objectives of this lesson are:

(a) To review the basic concepts of operating system i.e. definition, types, and functions performed by them.

(b) To review the process scheduling policies.

1. 1 Introduction

Operating System may be viewed as collection of software consisting of procedures for operating the computer and providing an environment for execution of programs. The main goals of the Operating System are:

(i) To make the computer system convenient to use,

(ii) To make the use of computer hardware in efficient way.

Basically, an Operating System has three main responsibilities:

(a) Perform basic tasks such as recognizing input from the keyboard, sending output to the display screen, keeping track of files and directories on the disk, and controlling peripheral devices such as disk drives and printers.

(b) Ensure that different programs and users running at the same time do not interfere with each other.

(c) Provide a software platform on top of which other programs can run.

1.2 Presentation of Contents

1.2.1 Operating System as a Resource Manager

- 1.2.1.1 Memory Management Functions
- 1.2.1.2 Processor/Process Management Functions
- 1.2.1.3 I/O Device Management Functions
- 1.2.1.4 Information Management Functions
- 1.2.1.5 Network Management Functions

1.2.2 Types of operating systems

1.2.2.1 Batch Operating System

1.2.2.2 Multiprogramming Operating System

1.2.2.3 Multitasking Operating System

1.2.2.4 Multi-user Operating System

1.2.2.5 Multithreading

1.2.2.6 Time-sharing system

1.2.2.7 Real-time systems

1.2.2.8 Combination of operating systems

1.2.2.9 Distributed Operating Systems

1.2.3 Process Scheduling

1.2.3.1 Definition of Process

1.2.3.2 Process States and Transitions

1.2.3.3 Types of schedulers

1.2.3.3.1 The long-term scheduler

1.2.3.3.2 The medium-term scheduler

1.2.3.3.3 The short-term scheduler

1.2.3.4 Scheduling and Performance Criteria

1.2.3.5 Scheduler Design

1.2.3.6 Scheduling Algorithms

1.2.3.6.1 First-Come, First-Served (FCFS) Scheduling

1.2.3.6.2 Shortest Job First (SJF)

1.2.3.6.3 Shortest Remaining Time Next (SRTN) Scheduling

1.2.3.6.4 Round Robin

1.2.3.6.5 Priority-Based Preemptive Scheduling (Event-Driven, ED)

1.2.3.6.6 Multiple-Level Queues (MLQ) Scheduling

1.2.3.6.7 Multiple-Level Queues with Feedback Scheduling

1.2.1 Operating System as a Resource Manager

We can view an Operating System as a resource manager that manages the resources of the computer system such as processor, memory, files storage space, input/output devices and so on which are to be required to solve a

computing problem. The Operating System allocates them to the specific programs and users as needed by their tasks. Since there can be many conflicting requests for the resources, the Operating System must decide which requests are to be allocated resources to operate the computer system fairly and efficiently. So the major functions of each category of Operating System are:

1.2.1.1 Memory Management Functions

To execute a program, it must be mapped to absolute addresses and loaded into memory. As the program executes, it accesses instructions and data from memory by generating these absolute addresses. The Operating System is responsible for the following memory management functions:

(i) Keep track of which segment of memory is in use and by whom.

(ii) Deciding which processes are to be loaded into memory, where it is to be loaded and how much when space becomes available.

(iii) Allocation or de-allocation the contents of memory when the process request for it otherwise reclaim the memory when the process does not require it or has been terminated.

1.2.1.2 Processor/Process Management Functions

A process is an instance of a program in execution. While a program is just a passive entity, process is an active entity performing the intended functions of its related program. A process needs certain resources like CPU, memory, files and I/O devices. In multiprogramming environment, there will a number of simultaneous processes existing in the system. The Operating System is responsible for the following processor/ process management functions:

(i) Provides mechanisms for process synchronization for sharing of resources amongst concurrent processes.

(ii) Keeps track of processor and status of processes.

(iii) Decide which process will be allocated the processor; the job scheduler chooses from all the submitted jobs and decides which one will be allowed into the system. If multiprogramming, decide which process gets the processor, when, for how much of time.

(iv) Allocate the processor to a process by setting up the necessary hardware registers. This module is widely known as the dispatcher.
(v) Providing mechanisms for deadlock handling.
(vi) Reclaim processor when process ceases to use a processor, or exceeds the allowed amount of usage.

1.2.1.3 I/O Device Management Functions

The Operating System is responsible for the following I/O Device Management Functions:

(i) Keep track of the I/O devices, I/O channels, etc.
(ii) Decide what is an efficient way to allocate the I/O resource. If it is to be shared, then decide who gets it, how much of it is to be allocated, and for how long. This is called I/O scheduling.
(iii) Allocate the I/O device and initiate the I/O operation.
(iv) Reclaim device as and when its use is through. In most cases I/O terminates automatically.

1.2.1.4 Information Management Functions

The major information management functions are:

(i) File system keeps track of the information, its location, its usage, status, etc.
(ii) Decides who gets hold of information, enforce protection mechanism, and provides for information access mechanism, etc.
(iii) Allocate the information to a requesting process, e.g., open a file.
(iv) De-allocate the resource, e.g., close a file.

1.2.1.5 Network Management Functions

An Operating System is responsible for the computer system networking via a distributed environment. A distributed system is a collection of processors, which do not share memory, clock pulse or any peripheral devices. Instead, each processor is having its own clock pulse, and RAM and they communicate through network. Access to shared resource permits increased speed, functionality and reliability.

1.2.2 TYPES OF OPERATING SYSTEMS

Operating System can be classified into various categories on the basis of several criteria, viz. number of simultaneously active programs, number of users working simultaneously, number of processors in the computer system, etc.

1.2.2.1 Batch Operating System

Batch processing requires the program, data, and appropriate system commands to be submitted together in the form of a job. Batch operating systems usually allow little interaction between users and executing programs. Batch processing has a greater potential for resource utilization than simple serial processing in computer systems serving multiple users. Due to turnaround delays and offline debugging, batch is not very convenient for program development. Programs that do not require interaction and need long execution times may be served well by a batch operating system. Memory management and scheduling in batch is very simple. Jobs are typically processed in first-come first-served fashion. Memory is usually divided into two areas. The resident portion of the Operating System permanently occupies one of them, and the other is used to load transient programs for execution. When a transient program terminates, a new program is loaded into the same area of memory. Since at most one program is in execution at any time, batch systems do not require any time-critical device management. Batch systems often provide simple forms of file management because of serial access to files. In it little protection and no concurrency control of file access is required.

1.2.2.2 Multiprogramming Operating System

A multiprogramming system permits multiple programs to be loaded into memory and execute the programs concurrently. Concurrent execution of programs results into improved system throughput and resource utilization. This potential is realized by a class of operating systems that multiplex resources of a computer system among a multitude of active programs. Such operating systems usually have the prefix multi in their names, such as multitasking or multiprogramming.

1.2.2.3 Multitasking Operating System

A multitasking Operating System is distinguished by its ability to support concurrent execution of two or more active processes. An instance of a program

in execution is called a process or a task. Multitasking is usually implemented by maintaining code and data of several processes in memory simultaneously, and by multiplexing processor and I/O devices among them. Multitasking is often coupled with hardware and software support for memory protection in order to prevent erroneous processes from corrupting address spaces and behavior of other resident processes. The terms multitasking and multiprocessing are often used interchangeably, although multiprocessing sometimes implies that more than one CPU is involved. In multitasking, only one CPU is involved, but it switches from one program to another so quickly that it gives the appearance of executing all of the programs at the same time. There are two basic types of multitasking: preemptive and cooperative. In preemptive multitasking, the Operating System parcels out CPU time slices to each program. In cooperative multitasking, each program can control the CPU for as long as it needs it. If a program is not using the CPU, however, it can allow another program to use it temporarily.

1.2.2.4 Multi-user Operating System

Multiprogramming operating systems usually support multiple users, in which case they are also called multi-user systems. Multi-user operating systems provide facilities for maintenance of individual user environments and therefore require user accounting. In general, multiprogramming implies multitasking, but multitasking does not imply multi-programming. In effect, multitasking operation is one of the mechanisms that a multiprogramming Operating System employs in managing the totality of computer-system resources, including processor, memory, and I/O devices. Multitasking operation without multi-user support can be found in operating systems of some advanced personal computers and in real-time systems. Multi-access operating systems allow simultaneous access to a computer system through two or more terminals. In general, multi-access operation does not necessarily imply multiprogramming. An example is provided by some dedicated transaction-processing systems, such as airline ticket reservation systems, that support hundreds of active terminals under control of a single program.

In general, the multiprocessing or multiprocessor operating systems manage the operation of computer systems that incorporate multiple processors. Multiprocessor operating systems are multitasking operating systems by definition because they support simultaneous execution of multiple tasks (processes) on different processors. Depending on implementation, multitasking may or may not be allowed on individual processors. Except for management and scheduling of multiple processors, multiprocessor operating systems provide the usual complement of other system services that may qualify them as time-sharing, real-time, or a combination operating system.

1.2.2.5 Multithreading

Multithreading allows different parts of a single program to run concurrently. The programmer must carefully design the program in such a way that all the threads can run at the same time without interfering with each other.

1.2.2.6 Time-sharing system

Time-sharing is a popular representative of multi-programmed, multi-user systems. One of the primary objectives of multi-user and time-sharing is good terminal response time. Time-sharing systems often attempt to provide equitable sharing of common resources, giving the illusion to each user of having a machine to oneself. Most time-sharing systems use time-slicing scheduling, in which programs are executed with rotating priority that increases during waiting and drops after the service is granted. In order to prevent programs from monopolizing the processor, a program executing longer than the system-defined time slice is interrupted by the Operating System and placed at the end of the queue of waiting programs. Memory management in time-sharing systems provides for isolation and protection of co-resident programs. Some forms of controlled sharing are sometimes provided to conserve memory and to exchange data between programs. Being executed on behalf of different users, programs in time-sharing systems generally do not have much need to communicate with each other. Allocation and de-allocation of devices must be done in a manner that preserves system integrity and provides for good performance.

1.2.2.7 Real-time systems

Real time systems are used in time critical environments where data must be processed extremely quickly because the output influences immediate decisions. A real time system must be responsive in time which is measured in fractions of seconds. In real time systems the correctness of the computations not only depends upon the logical correctness of the computation but also upon the time at which the results is produced. Real-time operating systems are used in environments where a large number of events, mostly external to the computer system, must be accepted and processed within certain deadlines.

A primary objective of real-time systems is to provide quick event-response times. User convenience and resource utilization are of secondary concern. It is not uncommon for a real-time system to be expected to process bursts of thousands of interrupts per second without missing a single event.

The Multitasking operation is accomplished by scheduling processes for execution independently of each other. Each process is assigned a certain level of priority that corresponds to the relative importance of the event that it services. The processor is normally allocated to the highest-priority process. Higher-priority processes usually preempt execution of the lower-priority processes. The process population in real-time systems is fairly static, and there is comparatively little moving of programs between primary and secondary storage. Processes in real-time systems tend to cooperate closely, thus necessitating support for both separation and sharing of memory. Time-critical device management is one of the main characteristics of real-time systems. In addition to providing sophisticated forms of interrupt management and I/O buffering, real-time operating systems often provide system calls to allow user processes to connect themselves to interrupt vectors and to service events directly. File management is found only in larger installations of real-time systems. In fact, some embedded real-time systems may not even have any secondary storage.

1.2.2.8 Combination of operating systems

Different types of Operating System are optimized to serve the needs of specific environments. In practice, however, a given environment may not exactly fit any of the described molds. For instance, both interactive program development and

lengthy simulations are often encountered in university computing centers. For this reason, some commercial operating systems provide a combination of described services. For example, a time-sharing system may support interactive users and also incorporate a full-fledged batch monitor. This allows computationally intensive non-interactive programs to be run concurrently with interactive programs. The common practice is to assign low priority to batch jobs and thus execute batched programs only when the processor would otherwise be idle. In other words, batch may be used as a filler to improve processor utilization while accomplishing a useful service of its own. Similarly, some time-critical events, such as receipt and transmission of network data packets, may be handled in real-time fashion on systems that otherwise provide time-sharing services to their terminal users.

1.2.2.9 Distributed Operating Systems

A distributed computer system is a collection of autonomous computer systems capable of communication and cooperation via their hardware and software interconnections. Distributed computer systems evolved from computer networks in which a number of largely independent hosts are connected by communication links and protocols. A distributed Operating System governs the operation of a distributed computer system and provides a virtual machine abstraction to its users. In distributed Operating System component and resource distribution should be hidden from users and application programs unless they explicitly demand. Distributed operating systems usually provide the means for system-wide sharing of resources, such as computational capacity, files, and I/O devices. In addition to typical operating-system services, a distributed Operating System may facilitate access to remote resources, communication with remote processes, and distribution of computations.

1.2.3 Process Scheduling

The most often requested resource is processor, so processor management is an important function carried out by the operating system.

One of the most fundamental concepts of modern operating systems is the distinction between a program and the activity of executing a program. The

former is merely a static set of directions; the latter is a dynamic activity whose properties change as time progresses. This activity is knows as a process. A process encompasses the current status of the activity, called the process state. This state includes the current position in the program being executed (the value of the program counter) as well as the values in the other CPU registers and the associated memory cells. The process state is a snapshot of the machine at that time. At different times during the execution of a program different snapshots will be observed.

The operating system allocates each process (a) a certain amount of time to use the processor, (b) various other resources that processes will need such as computer memory or disks. To keep track of the state of all the processes, the operating system maintains a table known as the process table. Inside this table, every process is listed along with the resources the processes are using and the current state of the process. Processes can be in one of three states: running, ready, or waiting (blocked). The running state means that the process has all the resources it need for execution and it has been given permission by the operating system to use the processor. Only one process can be in the running state at any given time. The remaining processes are either in a waiting state (i.e., waiting for some external event to occur such as user input or a disk access) or a ready state (i.e., waiting for permission to use the processor). The problem of determining when processors should be assigned and to which processes, is called processor scheduling or CPU scheduling. A scheduler is an Operating System module that selects the next job to be admitted into the system and the next process to run.

1.2.3.1 Definition of Process

The process has been given many definitions but the most frequently used definition is "*Process is a program in Execution*". In Process model, all software on the computer is organized into a number of sequential processes. The process state consist of everything necessary to resume the process execution if it is somehow put aside temporarily. The process state consists of at least following:

(i) Code for the program.
(ii) Program's static data.
(iii) Program's dynamic data.
(iv) Program's procedure call stack.
(v) Contents of general purpose register.
(vi) Contents of program counter (PC)
(vii) Contents of program status word (PSW).
(viii) Operating Systems resource in use.

A process goes through a series of discrete process states.

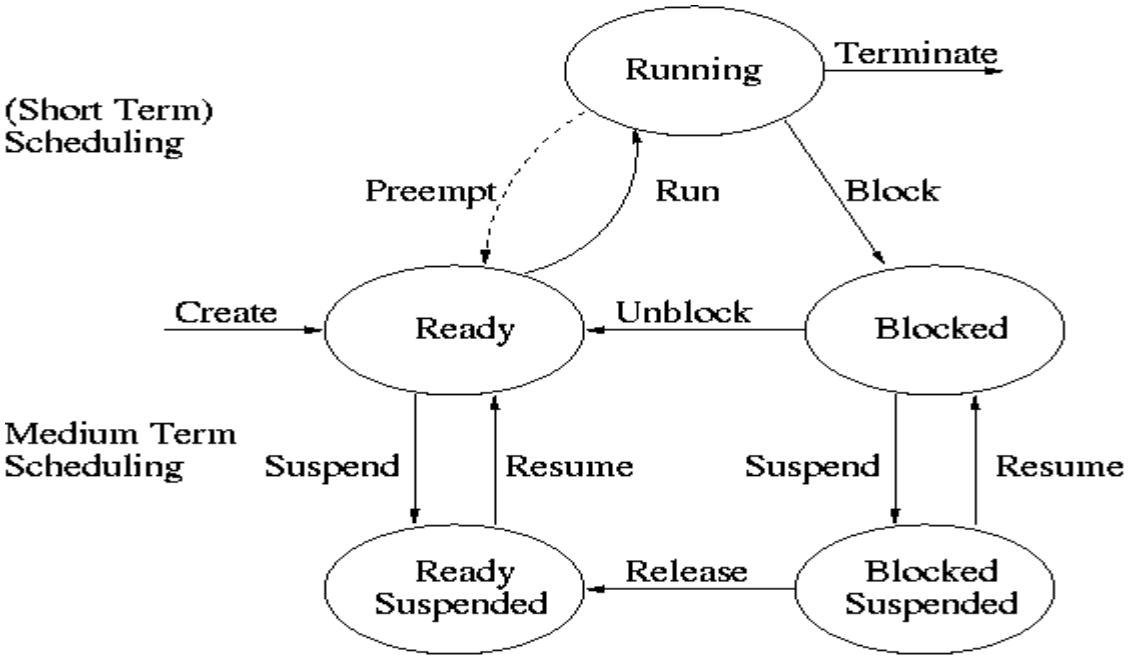
(a) **New State:** The process being created.
(b) **Running State:** A process is said to be running if process actually using the CPU at that particular instant.
(c) **Blocked State:** A process is said to be blocked if it is waiting for some event to happen before it can proceed.
(d) **Ready State:** A process is said to be ready if it use a CPU if one were available.
(e) **Terminated state:** The process has finished execution.

1.2.3.2 Process States and Transitions

The figure 1 contains much information. Consider a running process P that issues an I/O request. Then following events can take place:

(i) The process is blocked i.e. moved from running state to blocked state.
(ii) At some later point, a disk interrupt occurs and the driver detects that P's request is satisfied.
(iii) P is unblocked, i.e. is moved from blocked to ready
(iv) At some later time the operating system looks for a ready job to run and picks P and P moved to running state.

A suspended process (i.e. blocked) may be removed from the main memory and placed in the backup memory (blocked suspended). Subsequently they may be released and moved to the ready state by the medium term scheduler.



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

Figure 1

1.2.3.3 Types of schedulers

There are three distinct types of schedulers: a long-term scheduler, a mid-term or medium-term scheduler and a short-term scheduler. The names suggest the relative frequency with which these functions are performed. Figure 2 shows the possible traversal paths of jobs and programs through the components and queues, depicted by rectangles, of a computer system. The primary places of action of the three types of schedulers are marked with down-arrows. As shown in Figure 2, a submitted batch job joins the batch queue while waiting to be processed by the long-term scheduler. Once scheduled for execution, processes spawned by the batch job enter the ready queue to await processor allocation by the short-term scheduler. After becoming suspended, the running process may be removed from memory and swapped out to secondary storage. Such processes are subsequently admitted to main memory by the medium-term scheduler in order to be considered for execution by the short-term scheduler.

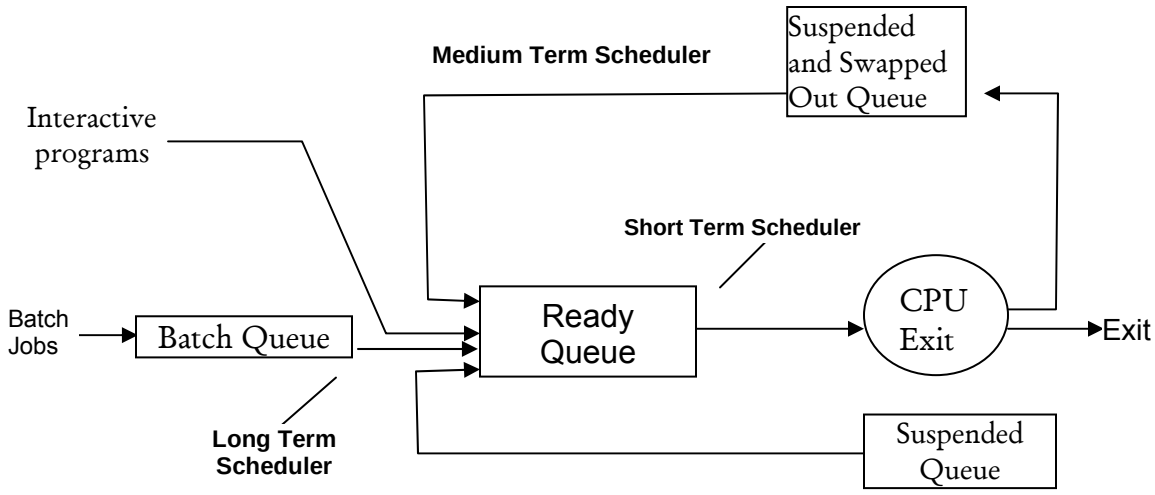


Figure 2- Process Schedulers

1.2.3.3.1 The long-term scheduler

The long-term scheduler decides when to start jobs. The long-term scheduler works with the batch queue and selects the next batch job to be executed. Batch is usually reserved for resource-intensive, low-priority programs that may be used as fillers to keep the system resources busy during periods of low activity of interactive jobs. Batch jobs contain all necessary data and commands for their execution. Batch jobs usually also contain programmer-assigned estimates of their resource needs. Knowledge about the anticipated job behavior facilitates the work of the long-term scheduler.

The primary objective of the long-term scheduler is to provide a balanced mix of jobs, such as processor-bound and I/O-bound, to the short-term scheduler so that processor utilization can be maximized. In addition, the long-term scheduler is usually invoked whenever a completed job departs the system. The frequency of invocation of the long-term scheduler is thus both system-and workload-dependent; but it is generally much lower than for the other two types of schedulers. As a result of the relatively infrequent execution and the availability of an estimate of its workload's characteristics, the long-term scheduler may incorporate rather complex and computationally intensive algorithms for admitting jobs into the system.

1.2.3.3.2 The medium-term scheduler

The medium term scheduler swaps out some process if memory is over-committed. The criteria for choosing a victim may be (a) How long since previously suspended? (b) How much CPU time used recently? (c) How much memory does it use? (d) External priority etc.

A running process may become suspended by making an I/O request or by issuing a system call. Given that suspended processes cannot make any progress towards completion until the related suspending condition is removed, it is sometimes beneficial to remove them from main memory to make room for other processes. In practice, the main-memory capacity may impose a limit on the number of active processes in the system. When a number of those processes become suspended, the remaining supply of ready processes in systems where all suspended processes remain resident in memory may become reduced to a level that impairs functioning of the short-term scheduler by leaving it few or no options for selection. In systems with no support for virtual memory, moving suspended processes to secondary storage may alleviate this problem.

The medium-term scheduler is in charge of handling the swapped-out processes. It has little to do while a process remains suspended. However, once the suspending condition is removed, the medium-term scheduler attempts to allocate the required amount of main memory, and swap the process in and make it ready.

So the medium-term scheduler controls suspended-to-ready transitions of swapped processes. This scheduler may be invoked when memory space is vacated by a departing process or when the supply of ready processes falls below a specified limit.

Medium-term scheduling is part of the swapping function of an operating system. The success of the medium-term scheduler is based on the degree of multiprogramming that it can maintain, by keeping as many processes “runnable” as possible. More processes can remain executable if we reduce the resident set size of all processes. The medium-term scheduler makes decisions as to which

pages of which processes need stay resident and which pages must be swapped out to make room for other processes. The sharing of some pages of memory, either explicitly or through the use of shared or dynamic link libraries complicates the task of the medium-term scheduler, which now must maintain reference counts on each page.

1.2.3.3.3 The short-term scheduler

The short-term scheduler (aka dispatcher), executes most frequently making decisions as to which process to move to Running next. The short-term scheduler is invoked whenever an event occurs which provides the opportunity of the interruption of the current process and the new (or continued) execution of another process. Such opportunities include:

(a) Clock interrupts, provide the opportunity to reschedule every few milliseconds,
(b) Expected I/O interrupts, when previous I/O requests are finally satisfied,
(c) Operating system calls, when the running process asks the operating system to perform an activity on its behalf, and
(d) Unexpected, asynchronous, events, such as unexpected input, user-interrupt, or a fault condition in the running program.

The short-term scheduler allocates the processor among the pool of ready processes resident in memory. Its main objective is to maximize system performance in accordance with the chosen set of criteria. Since it is in charge of ready-to-running state transitions, the short-term scheduler must be invoked for each process switch to select the next process to be run. In practice, the short-term scheduler is invoked whenever an event causes the global state of the system to change. Given that any such change could result in making the running process suspended or in making one or more suspended processes ready, the short-term scheduler should be run to determine whether such significant changes have indeed occurred and, if so, to select the next process to be run.

Most of the process-management Operating System services require invocation of the short-term scheduler as part of their processing. For example, creating a

process or resuming a suspended one adds another entry to the ready queue and the scheduler is invoked to determine whether the new entry should also become the running process. Suspending a running process, changing priority of the running process, and exiting or aborting a process are also events that may necessitate selection of a new running process.

As indicated in Figure 2, interactive programs often enter the ready queue directly after being submitted to the Operating System, which then creates the corresponding process. Unlike-batch jobs, the influx of interactive programs are not throttled, and they may conceivably saturate the system. The necessary control is usually provided indirectly by deterioration response time, which tempts the users to give up and try again later, or at least to reduce the rate of incoming requests.

Figure 2 illustrates the roles and the interplay among the various types of schedulers in an operating system. It depicts the most general case of all three types being present. For example, a larger operating system might support both batch and interactive programs and rely on swapping to maintain a well-behaved mix of active processes. Smaller or special-purpose operating systems may have only one or two types of schedulers available. Along-term scheduler is normally not found in systems without support for batch, and the medium-term scheduler is needed only when swapping is used by the underlying operating system. When more than one type of scheduler exists in an operating system, proper support for communication and interaction is very important for attaining satisfactory and balanced performance. For example, the long-term and the medium-term schedulers prepare workload for the short-term scheduler. If they do not provide a balanced mixed of compute-bound and I/O-bound processes, the short-term scheduler is not likely to perform well no matter how sophisticated it may be on its own merit.

1.2.3.4 Scheduling and Performance Criteria

The success of the short-term scheduler is evaluated against user-oriented criteria such as response time or system-oriented criteria such as throughput, the

rate at which tasks are completed. The following identifies some goals of a scheduling algorithm/policy according to the type of system.

All systems

(i) Fairness: all processes should be treated equitably.

(ii) Predictability: Regardless of the load on the system, it should be possible to estimate how long the job will take to complete.

(iii) Enforce priorities.

(iv) Avoid indefinite postponement: as a process waits for a resource its priority should grow (so-called aging).

(v) Degrade gracefully under heavy loads.

(vi) Balance: keep all parts of the system busy; processes using underutilized resources should be favored; give preference to processes holding key resources.

Batch systems

(i) Throughput: complete as many jobs as possible per unit time;

(ii) Turnaround: minimize time between submission of a job and its completion;

Interactive systems:

(i) Response times: maximize the number of interactive users receiving acceptable response times;

(ii) Be predictable: a given job should run in about the same amount of time regardless of the load on the system;

(iii) Give better service to processes exhibiting desirable behavior, e.g. low paging rates.

Real-time systems

(i) Meet deadlines.

(ii) Be predictable and degrade gracefully, e.g. if the going is getting really tough, retreat into a safety mode.

1.2.3.5 Scheduler design

Design process of a typical scheduler consists of selecting one or more primary performance criteria and ranking them in relative order of importance. The next step is to design a scheduling strategy that maximizes performance for the

specified set of criteria while obeying the design constraints. Schedulers typically attempt to maximize the average performance of a system, relative to a given criterion. However, due consideration must be given to controlling the variance and limiting the worst-case behavior.

One of the problems in selecting a set of performance criteria is that they often conflict with each other. For example, increased processor utilization is usually achieved by increasing the number of active processes, but then response time deteriorates. The design of a scheduler usually requires careful balance of all the different requirements and constraints. With the knowledge of the primary intended use of a given system, operating-system designers tend to maximize the criteria most important in a given environment.

1.2.3.6 Scheduling algorithms

The scheduling mechanisms described in this section may be used by any of the three types of schedulers although some algorithms are better suited to the needs of a particular type of scheduler. The scheduling policies may be categorized as preemptive and non-preemptive. Preemption means the operating system moves a process from running to ready without the process requesting it. Preemption is needed to guarantee fairness and it is found in all modern general-purpose operating systems.

Non-pre-emptive: In non-preemptive scheduling, once a process is executing, it will continue to execute until

(a) It terminates, or

(b) It makes an I/O request which would block the process, or

(c) It makes an operating system call.

Pre-emptive: In the preemptive scheduling, the same three conditions as above apply, and in addition the process may be pre-empted by the operating system when

(a) A new process arrives (perhaps at a higher priority), or

(b) An interrupt or signal occurs, or

(c) A (frequent) clock interrupt occurs.

CPU Scheduling deals with the problem of deciding which of the processes in the ready queue is to be allocated the CPU.

1.2.3.6.1 First-Come, First-Served (FCFS) Scheduling

The simplest selection function is the FCFS scheduling policy. In it

(i) The operating system kernel maintains all Ready processes in a single queue,

(ii) The process at the head of the queue is always selected to execute next,

(iii) The Running process runs to completion, unless it requests blocking I/O,

(iv) If the Running process blocks, it is placed at the end of the Ready queue.

Clearly, once a process commences execution, it will run as fast as possible (having 100% of the CPU, and being non-pre-emptive), but there are some obvious problems. By failing to take into consideration the state of the system and the resource requirements of the individual scheduling entities, FCFS scheduling may result in poor performance. As a consequence of no preemption, component utilization and the system throughput rate may be quite low.

Processes of short duration suffer when “stuck” behind very long-running processes because there is no discrimination on the basis of the required service. Compute-bound processes are favored over I/O-bound processes. We can measure the effect of FCFS by examining:

(i) The average turnaround time of each task (the sum of its waiting and running times), or

(ii) The normalized turnaround time (the ratio of running to waiting times).

1.2.3.6.2 Shortest Job First (SJF)

In this scheduling policy, the jobs are sorted on the basis of total execution time needed and then it run the shortest job first. It is a non-preemptive scheduling policy. Now first consider a static situation where all jobs are available in the beginning, and we know how long each one takes to run, and we implement “run-to-completion". In this situation, SJF has the shortest average waiting time. This scheduling policy can starve processes that require a long burst.

1.2.3.6.3 Shortest Remaining Time Next (SRTN) Scheduling

SRTN is a scheduling discipline in which the next process is selected on the basis of the shortest remaining execution time. SRTN scheduling may be implemented in either the non-preemptive or the preemptive variety. The non-preemptive version of SRTN is called shortest job first (SJF). In either case, whenever the SRTN scheduler is invoked, it searches the corresponding queue to find the process with the shortest remaining execution time. The difference between the two cases lies in the conditions that lead to invocation of the scheduler and, consequently, the frequency of its execution. Without preemption, the SRTN scheduler is invoked whenever a job is completed or the running process surrenders control to the Operating System. In the preemptive version, whenever an event occurs that makes a new process ready, the scheduler is invoked to compare the remaining processor execution time of the running process with the time needed to complete the next processor burst of the newcomer.

SRTN is optimal scheduling discipline in terms of minimizing the average waiting time of a given workload with a bias towards short jobs. With the addition of preemption, an SRTN scheduler can accommodate short jobs that arrive after commencement of a long job resulting in increased waiting times of long jobs.

The SRTN discipline schedules optimally assuming that the exact future execution times of processes are known at the time of scheduling. Dependence on future knowledge tends to limit the effectiveness of SRTN implementations.

Predictions of process execution requirements are usually based on observed past behavior, perhaps coupled with some other knowledge of the nature of the process and its long-term statistical properties, if available. A relatively simple predictor, called the exponential smoothing predictor, has the following form:

$$P_n = \alpha 0_n\text{-}1 + (1 - \alpha)P\text{-}1$$

where 0_n is the observed length of the (n-1)th execution interval, P_n-1 is the predictor for the same interval, and α is a number between 0 and 1. The parameter α controls the relative weight assigned to the past observations and predictions. For the extreme case of $\alpha = 1$, the past predictor is ignored, and the

new prediction equals the last observation. For $\alpha = 0$, the last observation is ignored. In general, expansion of the recursive relationship yields

$$P_n = \alpha \sum_{I=0}^{n-1} (1 - \alpha)^i 0_{n-i-1}$$

Thus the predictor includes the entire process history, with its more recent history weighted more.

Many operating systems measure and record elapsed execution time of a process in its PCB. This information is used for scheduling and accounting purposes. Implementation of SRTN scheduling obviously requires rather precise measurement and imposes the overhead of predictor calculation at run time. Moreover, some additional feedback mechanism is usually necessary for corrections when the predictor is grossly incorrect.

1.2.3.6.4 Round Robin

In interactive environments the primary requirement is to provide reasonably good response time and to share system resources equitably among all users. Obviously, only preemptive disciplines may be considered in such environments, and one of the most popular is time slicing, also known as round robin (RR). This preemptive scheduling policy gives each process a slice of time (i.e., one quantum) before being preempted. This scheduler works as follows:

- The processes that are ready to run are kept in a FIFO "Ready" queue.
- There is a fixed time quantum which any process runs at a time.
- The currently active process P runs until one of two things happens:
  - P blocks and put in the "blocked" state.
  - P exhausts its time quantum and pre-empted and put at the end of the ready queue.
- When a process unblocks it is put at the end of the ready queue.

The key parameter here is the quantum size q. Choice of the value of q is a tradeoff (1) Small q makes system more responsive, (2) Large q makes system more efficient since less process switching.

Round robin scheduling achieves equitable sharing of system resources. Short processes may be executed within a single time quantum and thus exhibit good response times. Long processes may require several quanta and thus be forced to cycle through the ready queue a few times before completion. With RR scheduling, response time of long processes is directly proportional to their resource requirements. For long processes that consist of a number of interactive sequences with the user, primarily the response time between the two consecutive interactions matters. If the computational requirements between two such sequences may be completed within a single time slice, the user should experience good response time. RR tends to subject long processes without interactive sequences to relatively long turnaround and waiting times. Such processes, however, may best be run in the batch mode, and it might even be desirable to discourage users from submitting them to the interactive scheduler.

RR can be tuned by adjusting q. If q is so high that it exceeds the overall time requirements for all processes, RR becomes the same as FCFS. As q tends to 0, process switching happens more frequently and eventually context switches occupy all available time. Thus q should be set small enough so that RR is fair but high enough so that the amount of time spent on context switching is reasonable.

1.2.3.6.5 Priority-Based Preemptive Scheduling (Event-Driven, ED)

In it each job is assigned a priority and the highest priority ready job is run and if many processes have the highest priority, it uses RR among them. Priorities may be static or dynamic. In either case, the user or the system assigns their initial values at the process-creating time. The level of priority may be determined as an aggregate figure on the basis of an initial value, characteristic, resource requirements, and run-time behavior of the process. In this sense, many scheduling disciplines may be regarded as being priority-driven, where the priority of a process represents its likelihood of being scheduled next. Priority-based scheduling may be preemptive or non-preemptive.

A common problem with this scheduling is the possibility that low-priority processes may be locked out by the higher priority ones and completion of a

process within finite time of its creation cannot be guaranteed but the usually remedy is provided by the aging priority.

Another variant of priority-based scheduling is used in hard real-time systems, where each process must be guaranteed execution before expiration of its deadline. Such processes are assumed to be assigned execution deadlines. The system workload consists of a combination of periodic processes, executed cyclically with a known period, and of periodic processes, whose arrival times are generally not predictable. An optimal scheduling discipline in such environments is the earliest-deadline scheduler, which schedules for execution the ready process with the earliest deadline. Another form of scheduler, called the least laxity scheduler has also been shown to be optimal in single-processor systems. This scheduler selects the ready process with the least difference between its deadline and computation time.

Priority aging

It is a solution to the problem of starvation. As a job is waiting, raise its priority so eventually it will have the maximum priority. This prevents starvation. It is preemptive policy. If there are many processes with the maximum priority, it uses FCFS among those with max priority (risks starvation if a job doesn't terminate) or can use RR.

1.2.3.6.6 Multiple-Level Queues (MLQ) Scheduling

The scheduling policies discussed so far are suited to particular applications. What should one use in a mixed system? A mix of scheduling disciplines may best service a mixed environment. For example, operating-system processes and device interrupts may be subjected to event-driven scheduling, interactive programs to round robin scheduling, and batch jobs to FCFS or STRN. One way to implement complex scheduling is to classify the workload according to its characteristics, and to maintain separate process queues serviced by different schedulers using an approach called multiple-level queues (MLQ) scheduling. A division of the workload might be into system processes, interactive programs, and batch jobs.

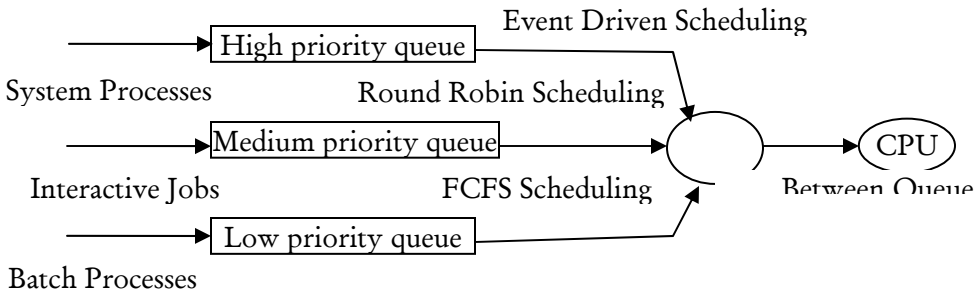


Figure 3 Multilevel Queue Scheduling

This would result in three ready queues, as depicted in Figure 3. A process may be assigned to a specific queue on the basis of its attributes. Each queue may then be serviced by the scheduling discipline best suited to the type of workload that it contains. Given a single server, some discipline must also be devised for scheduling between queues. Typical approaches are to use absolute priority or time slicing with some bias reflecting relative priority of the processes within specific queues. In the absolute priority case, the processes from the highest-priority queue are serviced until that queue becomes empty. The scheduling discipline may be event-driven, although FCFS should not be ruled out given its low overhead and the similar characteristics of processes in that queue. When the highest-priority queue becomes empty, the next queue may be serviced using its own scheduling discipline. Finally, when both higher-priority queues become empty, a batch-spawned process may be selected. A lower-priority process may, of course, be preempted by a higher-priority arrival in one of the upper-level queues. This discipline maintains responsiveness to external events and interrupts at the expense of frequent preemption's. An alternative approach is to assign a certain percentage of the processor time to each queue, commensurate with its priority.

1.2.3.6.7 Multiple-Level Queues with Feedback Scheduling

Multiple queues with feedback in a system may be used to increase the effectiveness and adaptive ness of scheduling. The idea here is to switch the queue of the processes on the basis of its run-time behavior. For example, each process may start at the top-level queue and if it is completed within a given time slice; it departs the system but if it need more than one time slice, it may be

reassigned to a lower-priority queue, which gets a lower percentage of the processor time. If the process is still not finished after having run a few times in that queue, it may be moved to yet another, lower-level queue. The idea is to give preferential treatment to short processes and have the resource-consuming ones slowly "sink" into lower-level queues, to be used as fillers to keep the processor utilization high.

On the other hand, if a process surrenders control to the Operating System before its time slice expires, it may be moved up in the hierarchy of queues to reward it. As in multiple-level queues, different queues may be serviced using different scheduling discipline but the introduction of feedback makes scheduling adaptive and responsive to the actual, measured run-time behavior of processes.

1.3 Summary

The prime responsibility of operating system is to manage the resources of the computer system. In addition to these, Operating System provides an interface between the user and the bare machine. On the basis of their attributes and design objectives, different types of operating systems were defined and characterized with respect to scheduling and management of memory, devices, and files. The primary concerns of a time-sharing system are equitable sharing of resources and responsiveness to interactive requests. Real-time operating systems are mostly concerned with responsive handling of external events generated by the controlled system. Distributed operating systems provide facilities for global naming and accessing of resources, for resource migration, and for distribution of computation. Process scheduling is a very important function of an operating system. Three different schedulers may coexist and interact in a complex operating system: long-term scheduler, medium-term scheduler, and short-term scheduler. Of the presented scheduling disciplines, FCFS scheduling is the easiest to implement but is a poor performer. SRTN scheduling is optimal but unrealizable. RR scheduling is most popular in time-sharing environments, and event-driven and earliest-deadline scheduling are dominant in real-time and other systems with time-critical requirements. Multiple-level queue scheduling, and its adaptive variant with feedback, is the most

general scheduling discipline suitable for complex environments that serve a mixture of processes with different characteristics.

1.4 Keywords

SPOOL: Simultaneous Peripheral Operations On Line

Task: An instance of a program in execution is called a process or a task.

Multitasking: The ability to execute more than one task at the same time is called as multitasking.

Real time: These systems are characterized by very quick processing of data because the output influences immediate decisions.

Multiprogramming: It is characterized by many programs simultaneously resident in memory, and execution switches between programs.

Long-term scheduling: the decisions to introduce new processes for execution, or re-execution.

Medium-term scheduling: the decision to add to (grow) the processes that are fully or partially in memory.

Short-term scheduling: It decides which (Ready) process to execute next.

Non-preemptive scheduling: In it, process will continue to execute until it terminates, or makes an I/O request which would block the process, or makes an operating system call.

Preemptive scheduling: In it, the process may be pre-empted by the operating system when a new process arrives (perhaps at a higher priority), or an interrupt or signal occurs, or a (frequent) clock interrupt occurs.

1.5 Self-Assessment Questions (SAQ)

1. What are the objectives of an operating system? Discuss.
2. Differentiate between multiprogramming, multitasking, and multiprocessing.
3. What are the major functions performed by an operating system? Explain.
4. Discuss various process scheduling policies with their cons and pros.
5. Define process. What are the different states of a process? Explain using a process state transition diagram.
6. What are the objectives of a good scheduling algorithm?

7. Explain the term context switch; how does the cost of a context switch affect the choice of scheduling algorithm.
8. From the point of view of scheduling, briefly explain the different requirements imposed by the following types of system: (a) batch, (b) interactive, (c) real-time.
9. Explain the need to compromise quantum used in round-robin.
10. Why round-robin scheduling is not suitable in batch operated computer system?

1.6 Suggested Readings / Reference Material

1. Operating System Concepts, 5th Edition, Silberschatz A., Galvin P.B., John Wiley and Sons.
2. Systems Programming and Operating Systems, 2nd Revised Edition, Dhamdhere D.M., Tata McGraw Hill Publishing Company Ltd., New Delhi.
3. Operating Systems, Madnick S.E., Donovan J.T., Tata McGraw Hill Publishing Company Ltd., New Delhi.
4. Operating Systems-A Modern Perspective, Gary Nutt, Pearson Education Asia, 2000.
5. Operating Systems, Harris J.A., Tata McGraw Hill Publishing Company Ltd., New Delhi, 2002.

Lesson Number: 2	***Writer: Dr. Rakesh Kumar***
Process Synchronization	***Vetter: Dr. Pradeep Bhatia***

2.0 Objectives

The objective of this lesson is get the students acquainted with the concepts of process synchronization. This lesson will make them familiar with:

(a) Critical section

(b) Mutual exclusion

(c) Classical coordination problems

2.1 Introduction

Since processes in concurrent systems frequently need to communicate with other processes therefore, there is a need for a well-structured communication, without using interrupts, among processes. Processes use two kinds of synchronization to control their activities;

(a) **Control synchronization:** it is needed if a process waits to perform some action only after some other processes have executed some action,

(b) **Data access synchronization:** It is used to access shared data in a mutually exclusive manner. The basic technique used to implement this synchronization is to block a process until an appropriate condition is fulfilled.

In this lesson synchronization in concurrent processes is discussed. Some classical coordination problems such as dining philosopher problem, producer-consumer problem etc are also discussed. These classical problems are the abstractions of the synchronization problems observed in operating systems.

2.2 Presentation of contents

2.2.1 Threads

Concurrency has many advantages and can be implemented by structuring an application as a set of concurrent processes. But a lot of overhead is involved in process switching and a low cost alternative to process is threads for achieving concurrency. Despite of the fact that a thread must execute in process, the process and its associated threads are different concept. Processes are used to group resources together and threads are the entities scheduled for execution on the CPU.

A thread is a single sequence stream within in a process. Because threads have some of the properties of processes, they are sometimes called lightweight processes. In a process, threads allow multiple executions of streams. In many respect, threads are popular way to improve application through parallelism. The CPU switches rapidly back and forth among the threads giving illusion that the threads are running in parallel. Like a traditional process i.e., process with one thread, a thread can be in any of several states (Running, Blocked, Ready or terminated). Each thread has its own stack. Since thread will generally call different procedures and thus a different execution history. This is why thread needs its own stack. An operating system that has thread facility, the basic unit of CPU utilization is a thread. A thread has or consists of a program counter (PC), a

register set, and a stack space. Threads are not independent of one other like processes as a result threads shares with other threads their code section, data section, Operating System resources also known as task, such as open files and signals.

2.2.1.1 Processes Vs Threads

Some of the similarities and differences between processes and threads are:

Similarities:

(i) Like processes threads share CPU and only one thread active at a time.
(ii) Like processes, threads within a process execute sequentially.
(iii) Like processes, thread can create children.
(iv) And like process, if one thread is blocked, another thread can run.

Differences

(i) Unlike processes, threads are not independent of one another.
(ii) Unlike processes, all threads can access every address in the task.
(iii) Unlike processes, threads are design to assist one other. Note that processes might or might not assist one another because processes may originate from different users.

2.2.1.2 Need of Threads

Following are some reasons why we use threads in designing operating systems:

(i) A process with multiple threads makes a great server for example printer server.
(ii) Because threads can share common data, they do not need to use interprocess communication.
(iii) Because of the very nature, threads can take advantage of multiprocessors.

Threads are cheap in the sense that

(i) They only need a stack and storage for registers therefore, threads are cheap to create.
(ii) Threads use very little resources of an operating system in which they are working. That is, threads do not need new address space, global data, program code or operating system resources.

(iii) Context switching is fast when working with threads. The reason is that we only have to save and/or restore PC, SP and registers.

But this cheapness does not come free - the biggest drawback is that there is no protection between threads.

2.2.1.3 User-Level Threads

User-level threads implement in user-level libraries, rather than via systems calls, so thread switching does not need to call operating system and to cause interrupt to the kernel. In fact, the kernel knows nothing about user-level threads and manages them as if they were single-threaded processes.

Advantages:

The most obvious advantage of this technique is that a user-level threads package can be implemented on an Operating System that does not support threads. Some other advantages are

(i) User-level threads do not require modification to operating systems.

(ii) Simple Representation:

Each thread is represented simply by a PC, registers, stack and a small control block, all stored in the user process address space.

- Simple Management: This simply means that creating a thread, switching between threads and synchronization between threads can all be done without intervention of the kernel.
- Fast and Efficient: Thread switching is not much more expensive than a procedure call.

Disadvantages:

(i) There is a lack of coordination between threads and operating system kernel. Therefore, process as whole gets one time slice irrespective of whether process has one thread or 1000 threads within. It is up to each thread to relinquish control to other threads.

(ii) User-level threads require non-blocking systems call i.e., a multithreaded kernel. Otherwise, entire process will blocked in the kernel, even if there are runable threads left in the processes. For example, if one thread causes a page fault, the process blocks.

2.2.1.4 Kernel-Level Threads

In this method, the kernel knows about and manages the threads. No runtime system is needed in this case. Instead of thread table in each process, the kernel has a thread table that keeps track of all threads in the system. In addition, the kernel also maintains the traditional process table to keep track of processes. Operating Systems kernel provides system call to create and manage threads.

Advantages:

(i) Because kernel has full knowledge of all threads, Scheduler may decide to give more time to a process having large number of threads than process having small number of threads.

(ii) Kernel-level threads are especially good for applications that frequently block.

Disadvantages:

(i) The kernel-level threads are slow and inefficient. For instance, threads operations are hundreds of times slower than that of user-level threads.

(ii) Since kernel must manage and schedule threads as well as processes. It requires a full thread control block (TCB) for each thread to maintain information about threads. As a result there is significant overhead and increased in kernel complexity.

2.2.1.5 Advantages of Threads over Multiple Processes

(i) **Context Switching:** Threads are very inexpensive to create and destroy, and they are inexpensive to represent. For example, they require space to store, the PC, the SP, and the general-purpose registers, but they do not require space to share memory information, Information about open files of I/O devices in use, etc. With so little context, it is much faster to switch between threads. In other words, it is relatively easier for a context switch using threads.

(ii) **Sharing:** Threads allow the sharing of a lot resources that cannot be shared in process, for example, sharing code section, data section, Operating System resources like open file etc.

2.2.1.6 Disadvantages of Threads over Multiprocesses

(i) **Blocking** The major disadvantage if that if the kernel is single threaded, a system call of one thread will block the whole process and CPU may be idle during the blocking period.

(ii) **Security** Since there is, an extensive sharing among threads there is a potential problem of security. It is quite possible that one thread over writes the stack of another thread (or damaged shared data) although it is very unlikely since threads are meant to cooperate on a single task.

2.2.1 Race Conditions

An atomic action is an indivisible action - an action taken by a process than cannot be interrupted by a context switch. When accesses to a shared variable are not atomic, a race condition may arise.

Consider the following extremely simple procedure

```
void deposit(int amount)
{
      balance = balance + amount;
 }
```

(Where we assume that balance is a shared variable). If two processes try to call deposit concurrently, something very bad can happen. Note that although balance = balance + amount looks like one statement but it is really implemented, on most computers, by a sequence of instructions such as

```
Load  reg, balance      //reg←balance
Add   reg, amount       //reg←reg + amount
Store reg, balance      //balane ← reg
```

Suppose process P1 calls deposit(10) and process P2 calls deposit(20). If one completes before the other starts, the combined effect is to add 30 to the balance, as desired. However, suppose the calls happen at exactly the same time, and the executions are interleaved. Suppose the initial balance is 100, and the two processes run on different CPUs. One possible result is

P1 loads 100 into its register

P2 loads 100 into its register

P1 adds 10 to its register, giving 110

P2 adds 20 to its register, giving 120

P1 stores 110 in balance

P2 stores 120 in balance

and the net effect is to add only 20 to the balance.

This kind of bug, which only occurs under certain timing conditions, is called a race condition. It is an extremely difficult kind of bug to track down (since it may disappear when you try to debug it) and may be nearly impossible to detect from testing (since it may occur only extremely rarely). The only way to deal with race conditions is through very careful coding. To avoid these kinds of problems, systems that support processes always contain constructs called synchronization primitives.

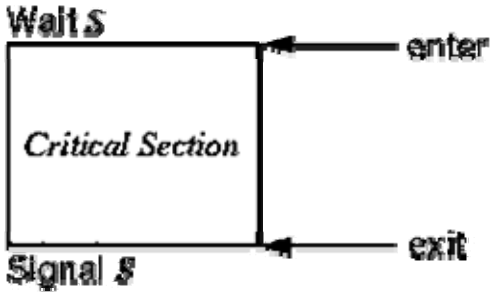
In operating systems, processes that are working together share some common storage (main memory, file etc.) that each process can read and write. When two or more processes are reading or writing some shared data and the final result depends on who runs precisely when, are called race conditions. Concurrently executing threads that share data need to synchronize their operations and processing in order to avoid race condition on shared data. Only one 'customer' thread at a time should be allowed to examine and update the shared variable. Race conditions are also possible in Operating Systems. If the ready queue is implemented as a linked list and if the ready queue is being manipulated during the handling of an interrupt, then interrupts must be disabled to prevent another interrupt before the first one completes. If interrupts are not disabled than the linked list could become corrupt.

So a race condition on a data item arises when many processes concurrently update its value. To maintain consistency, any time only one process should update the value. How to avoid race conditions? One solution is critical section.

2.2.2 Critical Section

The key to preventing trouble involving shared storage is find some way to prohibit more than one process from reading and writing the shared data simultaneously. Not all code needs to worry about race conditions. Only code

that is manipulating shared data needs to worry about them. The most common way to avoid race conditions is to mark the sections of code that are accessing shared data (That part of the program where the shared memory is accessed is called the Critical Section or critical regions) and insure that only one process is ever executing code in the critical section or context switched out of code in the critical section. This property is called mutual exclusion. To avoid race conditions and flawed results, one must identify codes in Critical Sections in each thread.



The characteristic properties of the code that form a Critical Section are

(i) Codes that reference one or more variables in a “read-update-write” fashion while any of those variables is possibly being altered by another thread.

(ii) Codes that alter one or more variables that are possibly being referenced in “read-update-write” fashion by another thread.

(iii) Codes use a data structure while any part of it is possibly being altered by another thread.

(iv) Codes alter any part of a data structure while it is possibly in use by another thread.

Here, the important point is that when one process is executing shared modifiable data in its critical section, no other process is to be allowed to execute in its critical section. Thus, the execution of critical sections by the processes is mutually exclusive in time. So a critical section for a data item d is defined as a section of code, which cannot be executed concurrently with itself or with other critical sections of d.

Consider a system consisting of n processes $\{P_0, P_1, ..., P_{n-i}\}$. Each process has a segment of code, called a critical section, in which the process may be

changing common variables, updating a table, writing a file, and so on. The important feature of the system is that, when one process is executing its critical section, no other process is to be allowed to execute in its critical section. Thus, the execution of critical sections by the processes is mutually exclusive in time. The critical-section problem is to design a protocol that the processes can use to co-operate. Each process must request permission to enter its critical section. The section of code implementing this request is the entry section. An exit section may follow the critical section. The remaining code is the remainder section.

Repeat

Entry Section

Critical section

Exit Section

Remainder section

Until FALSE

Properties of critical section

The following properties are to be possessed by an implementation of critical section:

(i) **Correctness:** At most one process may execute a critical section at any given moment.

(ii) **Progress:** When a critical section is not in use, one of the processes visiting to enter it will be granted entry to the critical section. If no process is executing in its critical section and there exist some processes that wish to enter their critical sections, then only those processes that are not executing in their remainder section can participate in the decision of which will enter its critical section next. Moreover, this decision cannot be postponed indefinitely. So if no process is in critical section, one can decide quickly who enters and only one process can enter the critical section so in practice, others are put on the queue.

(iii) **Bounded wait:** After a process p has indicated its desire to enter a critical section, the number of times other processes gain entry to the critical section ahead of p is bounded by a finite integer. So there must exist a bound on the number of times that other processes are allowed to enter their critical sections after a process has made a

request to enter its critical section and before that request is granted. The wait is the time from when a process makes a request to enter its critical section until that request is granted. In practice, once a process enters its critical section, it does not get another turn until a waiting process gets a turn (managed as a queue)

(iv) **Deadlock freedom:** The implementation is free of deadlock.

2.2.3 Mutual Exclusion

It is a way of making sure that if one process is using a shared modifiable data, the other processes will be excluded from doing the same thing.

Formally, while one process executes the shared variable, all other processes desiring to do so at the same time moment should be kept waiting; when that process has finished executing the shared variable, one of the processes waiting to do so should be allowed to proceed. In this fashion, each process executing the shared data (variables) excludes all others from doing so simultaneously. This is called Mutual Exclusion.

Note that mutual exclusion needs to be enforced only when processes access shared modifiable data - when processes are performing operations that do not conflict with one another they should be allowed to proceed concurrently.

2.2.3.1 Mutual Exclusion Conditions

If we could arrange matters such that no two processes were ever in their critical sections simultaneously, we could avoid race conditions. We need four conditions to hold to have a good solution for the critical section problem (mutual exclusion).

(i) No two processes may at the same moment inside their critical sections.

(ii) No assumptions are made about relative speeds of processes or number of CPUs.

(iii) No process outside its critical section should block other processes.

(iv) No process should wait arbitrary long to enter its critical section.

2.2.3.2 Proposals for Achieving Mutual Exclusion

The mutual exclusion problem is to devise a pre-protocol (or entry protocol) and a post-protocol (or exist protocol) to keep two or more threads from being in their critical sections at the same time.

Problem

When one process is updating shared modifiable data in its critical section, no other process should allow entering in its critical section.

Proposal 1 -Disabling Interrupts (Hardware Solution)

Each process disables all interrupts just after entering in its critical section and re-enable all interrupts just before leaving critical section. With interrupts turned off the CPU could not be switched to other process. Hence, no other process will enter its critical section and mutual exclusion achieved.

Disabling interrupts is sometimes a useful technique within the kernel of an operating system, but it is not appropriate as a general mutual exclusion mechanism for users process. The reason is that it is unwise to give user process the power to turn off interrupts. It's a bad idea for several reasons:

(i) It badly breaks process isolation - a process can prevent other processes from hearing from the disk or timers.

(ii) Turning off interrupts for any period of time is extremely hazardous. The Operating System needs to field interrupts to keep the machine running, and frequently will be so confused after interrupts are off for a while that it will reboot anyway. (Most hardware will send an NMI (non maskable interrupt) after too long with interrupts disabled).

Proposal 2 - Lock Variable (Software Solution)

In this solution, we consider a single, shared, (lock) variable, initially 0. When a process wants to enter in its critical section, it first tests the lock. If lock is 0, the process first sets it to 1 and then enters the critical section. If the lock is already 1, the process just waits until (lock) variable becomes 0. Thus, a 0 means that no process in its critical section, and 1 means hold your horses - some process is in its critical section.

```
Int lock;
Repeat
void enter_region (int process)
{
        while ( lock == 1 ); // do nothing
```

```
        lock = 1;
}
{       Critical section        }
void leave_region(int process)
{
        lock = 0;
}
{       Remainder Section  }
Forever
```

The flaw in this proposal can be best explained by example. Suppose process A sees that the lock is 0. Before it can set the lock to 1 another process B is scheduled, runs, and sets the lock to 1. When the process A runs again, it will also set the lock to 1, and two processes will be in their critical section simultaneously. The code fails because lock is as much a shared variable as anything in the critical section. The same way that two processes increment balance variable in our first example, two processes can both see the lock as zero before it is set to one, and both enter the critical section.

Proposal 3 - Strict Alteration

In this proposed solution, the integer variable 'turn' keeps track of whose turn is to enter the critical section. Initially, process A inspects turn, finds it to be 1, and enters in its critical section. Process B also finds it to be 1 and sits in a loop continually testing 'turn' to see when it becomes 2. Continuously testing a variable waiting for some value to appear is called the Busy-Waiting.

Algorithm

```
Var turn: integer
Begin
turn=1
Concurrent begin
Repeat                  Repeat
  While turn=2            While turn=1
     Do {nothing}            Do {nothing}
```

{critical section}	{critical section}
turn=2	turn=1
{remainder section}	{remainder section}
forever	forever
concurrent end	
end.	
Process p1	**Process p2**

The shared variable turn is used to indicate which process can enter the critical section next. Let process p1 wish to enter the Critical Section. If turn=1, p1 can enter straightway. After completing the Critical Section, it sets turn to 2 so as to enable process p2 to enter the Critical Section. If p1 finds turn=2 when it wishes to enter Critical Section, it waits in the while loop until p2 exits from the critical section and executes the assignment turn=1. Thus processes may encounter a busy wait before gaining entry to the Critical Section.

Taking turns is not a good idea when one of the processes is much slower than the other. Suppose process 1 finishes its critical section quickly, so both processes are now in their non-critical section. This situation violates above-mentioned condition 3(i.e. No process outside its critical section should block other processes.). E.g. let process p1 be in critical section and p2 in the remainder section. If p1 exits from the Critical Section, finishes the remainder section and wishes to enter the critical section again, it will face busy wait until after p2 uses the Critical Section. So the progress condition is violated since p1 is not able to enter although critical section is available. The solution of this problem can be:

Var c1, c2: integer	
C1=1;c2=1;	
{	
repeat	Repeat
while c2=0 do {nothing}	while c1=0 do {nothing}
c1=0;	c2=0;
{critical section}	{critical section}

c1=1;
{remainder section}
forever
}
	c2=1;
	{remainder section}
	forever
	}

Process p1	**Process p2**

In this algorithm, c1 is a status flag for p1. p1 sets this flag to 0 while entering the critical section and change it to 1 upon exit. P2 checks the status of c1, if it is 1 then enter otherwise wait. This check eliminates the progress violation by enabling a process to enter critical section again any number of times if other process is not interested in entering the Critical Section. However the busy wait condition still exists.

Using Systems calls 'sleep' and 'wakeup'

Basically, what above-mentioned solution does is this: when a process wants to enter in its critical section, it checks to see if the entry is allowed. If it is not, then it waits until it is allowed to enter. This approach waste CPU-time.

Now look at some interprocess communication primitives i.e. the pair of steep-wakeup.

(i) **Sleep:** It is a system call that causes the caller to block, that is, be suspended until some other process wakes it up.

(ii) **Wakeup:** It is a system call that wakes up the process.

Both 'sleep' and 'wakeup' system calls have one parameter that represents a memory address used to match up 'sleeps' and 'wakeups'.

2.2.4 Classical process co-ordination problems

There are a number of different process co-ordination problems arising in practical situations that exemplify important associated issues. These problems also provide base for solution testing for process co-ordination problems. In this section, we will see some of such classical process co-ordination problems.

2.2.4.1 The readers and writers problem

Courtois, Heymans, and Pumas posed an interesting synchronization problem called the readers-writers problem. Suppose a resource is to be shared among a community of processes of two distinct types: readers and writers. A reader

process can share the resource with any other reader process but not with any writer process. A writer process acquires exclusive access to the resource whenever it requires any access to the resource.

This scenario is similar to one in which a file is to be shared among a set of processes, If a process wants only to read the file, then it may share the file with any other process that also wants to read the file. If a writer wants to modify the file, then no other process should have access to the file writer has access to it.

The correctness conditions are as follows:

1. There are two classes of processes:
    (a) Readers, which can work concurrently.
    (b) Writers, which need exclusive access.
2. Only one writer can write any time. So we must prevent 2 writers from being concurrent.
3. We must prevent a reader and a writer from being concurrent. So reading is prohibited while a writer is writing.
4. We must permit readers to be concurrent when no writer is active. So many readers can read simultaneously.
5. Perhaps we want fairness (i.e., freedom from starvation).
6. Optional variants can be:
    a. Writer should have priority over readers.
    b. Readers should have priority over writers.

The “easy way out" is to treat all processes as writers in which case the problem reduces to mutual exclusion. The disadvantage of the easy way out is that you give up reader concurrency. A possible solution is as under:

```
{                                   {
repeat                              repeat
        if a writer is writing              if readers reading or writer is writing
        then {wait}                         then {wait}
```

{read}	{write}
If writer waiting	If reader or writer waiting
Then	Then
Wake a writer if no Readers are reading.	Wake all readers Or one writer
forever	forever
}	}
Process Readers	**Process Writers**

2.2.4.2 The Bounded Buffer Producers and Consumers

As an example how sleep-wakeup system calls are used, consider the producer-consumer problem also known as bounded buffer problem. In the producers and consumers problem, there are two classes of processes

(i) Producers, which produce items and insert them into a buffer.

(ii) Consumers, which remove items and consume them.

These two processes share a common, fixed-size (bounded) buffer. The producer puts information into the buffer and the consumer takes information out.

Trouble arises when

1. The producer wants to put a new data in the buffer, but buffer is already full.
   Solution: Producer goes to sleep and to be awakened when the consumer has removed data.
2. The consumer wants to remove data from the buffer but buffer is already empty.
   Solution: Consumer goes to sleep until the producer puts some data in buffer and wakes consumer up.

The bounded buffer producers and consumers assume that there is a fixed buffer size i.e., a finite numbers of slots is available.

This approach also leads to same race conditions we have seen in earlier approaches. Race condition can occur due to the fact that access to 'count' is

unconstrained. The essence of the problem is that a wakeup call, sent to a process that is not sleeping, is lost.

Another solution is the use of monitors. Monitors is a high-level data abstraction tool that automatically generates atomic operations on a given data structure. A monitor has:

(i) Shared data.
(ii) A set of atomic operations on that data.
(iii) A set of condition variables.

Each monitor has one lock. It acquires lock when begin a monitor operation, and releases lock when operation finishes. It statically identifies operations that only read data, and then allow these read-only operations to go concurrently. Writers get mutual exclusion with respect to other writers and to readers. The advantages of using monitors are (i) it reduces probability of error (never forget to Acquire or Release the lock), (ii) biases programmer to think about the system in a certain way (is not ideologically neutral).

Bounded buffer using monitors and signals

(i) **Shared State** data [num] - a buffer holding produced data. num - tells how many produced data items there are in the buffer.
(ii) **Atomic Operations** Produce (v) called when producer produces data item v. Consume (v) called when consumer is ready to consume a data item. Consumed item put into v.
(iii) **Condition Variables** There are two condition variables (1) bufferAvail - signalled when a buffer becomes available. (2) dataAvail - signalled when data becomes available.

```
Condition *bufferAvail, *dataAvail;
int num = 0;
int data[10];
Lock *monitorLock;
Produce (v)
{
monitorLock->Acquire(); /* Acquire monitor lock - makes operation atomic */
```

```
  while (num == 10)
        {  bufferAvail->Wait(monitorLock);  }
   put v into data array
  num++;
   dataAvail->Signal(monitorLock);
    monitorLock->Release(); /* Release monitor lock after perform operation */
  }
 Consume(v)
{
  monitorLock->Acquire(); /* Acquire monitor lock - makes operation atomic */
  while (num == 0)
     { dataAvail->Wait(monitorLock); }
  put next data array value into v
  num--;
  bufferAvail->Signal(monitorLock);
  monitorLock->Release(); /* Release monitor lock after perform operation */
}
```

2.2.5 Semaphores

E.W. Dijkstra (1965) abstracted the key notion of mutual exclusion in his concepts of semaphores.

Definition

A semaphore is a protected variable whose value can be accessed and altered only by the operations P and V and initialization operation called 'Semaphoiinitislize'. Binary Semaphores can assume only the value 0 or the value 1, counting semaphores, also called general semaphores, can assume only nonnegative values.

P() or down() decrements the semaphore by one. If the semaphore is zero, the process calling P() is blocked until the semaphore is positive again. The P (or wait or sleep or down) operation on semaphores S, written as P(S) or wait (S), operates as follows:

P(S): IF S >0

THEN S = S – 1

ELSE (wait on S)

V() or up() increments the semaphore by one. The V (or signal or wakeup or up) operation on semaphore S, written as V(S) or signal (S), operates as follows:

V(S): IF (one or more process are waiting on S)

THEN (let one of these processes proceed)

ELSE S = S + 1

Operations P and V are done as single, indivisible, atomic action. It is guaranteed that once a semaphore operation has stared, no other process can access the semaphore until operation has completed. Mutual exclusion on the semaphore, S, is enforced within P(S) and V(S).

If several processes attempt a P(S) simultaneously, only one process will be allowed to proceed. The other processes will be kept waiting, but the implementation of P and V guarantees that processes will not suffer indefinite postponement.

Semaphores solve the lost-wakeup problem.

2.2.5.1 Producer-Consumer Problem Using Semaphores

The Solution to producer-consumer problem uses three semaphores namely, full, empty and mutex. The semaphore 'full' is used for counting the number of slots in the buffer that are full. The 'empty' for counting the number of slots that are empty and semaphore 'mutex' to make sure that the producer and consumer do not access modifiable shared section of the buffer simultaneously.

Initialization

(i) Set full buffer slots to 0 i.e., semaphore Full = 0.

(ii) Set empty buffer slots to N i.e., semaphore empty = N.

(iii) For control access to critical section set mutex to 1 i.e., semaphore mutex=1.

```
Producer ( )
WHILE (true)
        produce-Item ( );
    P (empty);
    P (mutex);
```

```
        enter-Item ( )
        V (mutex)
        V (full);
Consumer ( )
WHILE (true)
        P (full)
        P (mutex);
        remove-Item ( );
        V (mutex);
        V (empty);
        consume-Item (Item)
```

2.2.6 The Dining philosophers' problem

The dining philosophers' problem is a "classical" synchronization problem. Taken at face value, it is a pretty meaningless problem, but it is typical of many synchronization problems that you will see when allocating resources in operating systems.

The problem is defined as follows: There are 5 philosophers sitting at a round table. Between each adjacent pair of philosophers is a chopstick. In other words, there are five chopsticks. Each philosopher does two things: think and eat. The philosopher thinks for a while, and then stops thinking and becomes hungry. When the philosopher becomes hungry, he/she cannot eat until he/she owns the chopsticks to his/her left and right. When the philosopher is done eating he/she puts down the chopsticks and begins thinking again.

Why describe problems this way? Well, the analogous situations in computers are sometimes so technical that they obscure creative thought. Thinking about philosophers makes it easier to think abstractly. And many of the early students of this field were theoreticians who like abstract problems. There are a bunch of named problems - Dining Philosophers, Drinking Philiosophers, Byzantine Generals, etc.

The challenge in the dining philosophers problem is to design a protocol so that the philosophers do not deadlock (i.e. every philosopher has a chopstick), and so

that no philosopher starves (i.e. when a philosopher is hungry, he/she eventually gets the chopsticks). Additionally, our protocol should try to be as efficient as possible -- in other words, we should try to minimize the time that philosophers spent waiting to eat.

A simple solution to this problem can be of the form

Repeat

Lift the left fork

Lift the right fork

{Eat}

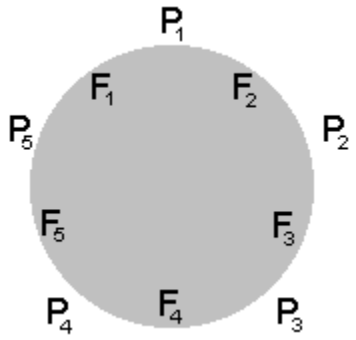
Put the left fork

Put the right fork

{Think}

Forever

But this solution is not acceptable since it is prone to deadlock (If all the philosophers lift their left fork).



One solution is to order the forks and require the philosophers to pick up the forks in increasing order, which mathematically eliminates the possibility of a deadlock. To illustrate this solution, label the philosophers P_1, P_2, P_3, P_4, and P_5, and label the forks F_1, F_2, F_3, F_4, and F_5. Each philosopher must pick up forks in a prescribed order and cannot pick up a fork another philosopher already has. Upon acquiring two forks, a philosopher may eat. Philosophers P_1 through P_4 follow the rule that P_x must pick up fork F_x first and then may pick up fork F_{x+1}. For example, P_1 must pick up F_1 first and F_2 second. Philosopher P_5 must,

conversely, pick up fork F_1 before picking up fork F_5, to respect the deadlock-preventing fork ordering rule.

Although avoiding a deadlock, this solution is inefficient, because one can arrive to a situation where only one philosopher is eating and everybody else is waiting for him. For example philosophers P_1 to P_3 could hold forks F_1 to F_3, waiting to get forks F_2 to F_4 respectively, philosopher P_5 could wait on fork F_1 having no fork yet, while philosopher P_4 would be eating holding forks F_4 and F_5. Optimally, either philosopher P_1 or philosopher P_2 should be able to eat in such circumstances.

Preventing starvation depends on the method of mutual exclusion enforcement used. Implementations using spinlocks or busy waiting can cause starvation through timing problems inherent in these methods. Other methods of mutual exclusion that utilize queues can prevent starvation by enforcing equal access to a fork by the adjacent philosophers.

2.3 Summary

In operating systems, concurrent processes share some common storage that each process can read and write. Since processes frequently need to communicate with other processes therefore, there is a need for a well-structured communication. Processes use two kinds of synchronization to control their activities; Control synchronization, & Data access synchronization. A race condition on a data item arises when many processes concurrently update its value. One solution to race condition is critical section. If we could arrange matters such that no two processes were ever in their critical sections simultaneously, we could avoid race conditions. There are four conditions to hold to have a good solution for the critical section problem (mutual exclusion). In the approaches to mutual exclusion, if a process wants to enter in its critical section, it checks to see if the entry is allowed. If it is not allowed to enter, it waits. This approach waste CPU-time. To avoid this sleep and wakeup calls are used.

A semaphore is a protected variable whose value can be accessed and altered only by the indivisible operations P and V and initialization operation called 'Semaphoiinitislize' Semaphores helps in avoiding the race condition

2.4 Keywords

Threads: A thread is a single sequence stream within in a process & they are sometimes called lightweight processes.

Critical section: that part of the program where the shared memory is accessed.

Mutual Exclusion: each process executing the shared data excludes all others from doing so simultaneously.

Race condition: When two or more processes are reading or writing some shared data and the final result depends on who runs precisely when, are called race conditions.

Semaphore: an object that hides an integer value and only allows three operations: initialization to a specified value, increment, or decrement.

2.5 Self assessment questions

1. What is a thread? Differentiate between user level threads and kernel level threads.
2. What is the difference between a thread and a process? Discuss the merits/demerits of threads over processes.
3. What is a race condition? Explain using a suitable example.
4. What do you understand by critical section? What are the charcteristic properties of it? Explain.
5. What is mutual exclusion? Discuss the different approaches to solve the problem of mutual exclusion.
6. What do you understand by semphores? Does it satisfy the bounded wait condition? Explain.
7. What is semaphore? How does it help in avoiding the race condition? Explain.
8. What are the limitations of hardware solutions for achieving mutual exclusion?
9. What do you understand by: (a) Busy waiting, (b) Bounded wait

2.6 Suggested readings/reference material

6. Operating System Concepts, 5th Edition, Silberschatz A., Galvin P.B., John Wiley & Sons.
7. Systems Programming & Operating Systems, 2nd Revised Edition, Dhamdhere D.M., Tata McGraw Hill Publishing Company Ltd., New Delhi.

8. Operating Systems, Madnick S.E., Donovan J.T., Tata McGraw Hill Publishing Company Ltd., New Delhi.
9. Operating Systems-A Modern Perspective, Gary Nutt, Pearson Education Asia, 2000.
10. Operating Systems, Harris J.A., Tata McGraw Hill Publishing Company Ltd., New Delhi, 2002

Lesson number: 3	Writer: Dr. Rakesh Kumar
Deadlocks	***Vetter: Dr. Pradeep Bhatia***

3.0 Objectives

The objectives of this lesson are to make the students acquainted with the problem of deadlocks. In this lesson, we characterize the problem of deadlocks and discuss policies, which an Operating System can use to ensure their absence. Deadlock detection, resolution, prevention and avoidance have been discussed in detail in the present lesson.

After studying this lesson the students will be familiar with following:

(a) Condition for deadlock.

(b) Deadlock prevention

(c) Deadlock avoidance

(d) Deadlock detection and recovery

3.1 Introduction

A serious danger of concurrent programming is deadlock. A race condition produces incorrect results, where a deadlock results in the deadlocked processes never making any progress. In the simplest form it's a process waiting for a resource held by a second process that's waiting for a resource that the first holds.

In a multiprogramming environment where several processes compete for resources, a situation may arise where a process is waiting for resources that are held by other waiting processes. This situation is called a deadlock. Generally, a system has a finite set of resources (such as memory, IO devices, etc.) and a finite set of processes that need to use these resources. A process which wishes to use any of these resources makes a request to use that resource. If the resource is free, the process gets it. If it is used by another process, it waits for it to become free. The assumption is that the resource will eventually become free and the waiting process will continue on to use the resource. But what if the other process is also waiting for some resource?

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

If a process is in the need of some resource, physical or logical, it requests the kernel of operating system. The kernel, being the resource manager, allocates the resources to the processes. If there is a delay in the allocation of the resource to the process, it results in the idling of process. The deadlock is a situation in which some processes in the system faces indefinite delays in resource allocation. In this lesson, we identify the problems causing deadlocks, and discuss a number of policies used by the operating system to deal with the problem of deadlocks.

3.2 Presentation of contents

3.2.1 Definition

3.2.2 Preemptable and Nonpreemptable Resources

3.2.3 Necessary and Sufficient Deadlock Conditions

3.2.4 Resource-Allocation Graph

3.2.4.1 Interpreting a Resource Allocation Graph with Single Resource Instances

3.2.5 Dealing with Deadlock

3.2.6 Deadlock Prevention

3.2.6.1 Elimination of “Mutual Exclusion” Condition

3.2.6.2 Elimination of “Hold and Wait” Condition

3.2.6.3 Elimination of “No-preemption” Condition

3.2.6.4 Elimination of “Circular Wait” Condition

3.2.7 Deadlock Avoidance

3.2.7.1 Banker’s Algorithm

3.2.7.2 Evaluation of Deadlock Avoidance Using the Banker’s Algorithm

3.2.8 Deadlock Detection

3.2.9 Deadlock Recovery

3.2.10 Mixed approaches to deadlock handling

3.2.11 Evaluating the Approaches to Dealing with Deadlock

3.2.1 Definition

A deadlock involving a set of processes D is a situation in which:

(a) Every process P_i in D is blocked on some event E_i.

(b) Event E_i can be caused only by action of some process (es) in D.

In other words, each member of the set of deadlock processes is waiting for a resource that can be released only by a deadlock process. None of the processes can run, none of them can release any resources, and none of them can be awakened. The resources may be either physical or logical. Examples of physical resources are Printers, Tape Drivers, Memory Space, and CPU Cycles. Examples of logical resources are Files, Semaphores, and Monitors.

Here's an example:

```
#define N 100 /* Number of free slots */
semaphore mutex = 1;
semaphore empty = N;
semaphore full = 0;
void producer()
{
        item i;
        while (1)
        {
                i = produce_item();
                P (mutex);
                P (empty);
                insert_new_item(i);
                V (mutex);
                V (full);
        }
}
void consumer()
{
        item i;
        while (1)
```

```
        {
                P (full);
                P (mutex);
                i = get_next_item();
                V (mutex);
                V (empty);
                consume_item (i);
        }
}
```

When the producer arrives at a full queue, it will block on empty while holding mutex, and the consumer will subsequently block on mutex forever.

What are the consequences of deadlocks?

- Response times and elapsed times of processes suffer.
- If a process is allocated a resource R1 that it is not using and if some other process P2 requires the resource, then P2 is denied the resource and the resource remains idle.

3.2.2 Preemptable and Nonpreemptable Resources

Resources come in two flavors: preemptable and nonpreemptable. A preemptable resource is one that can be taken away from the process with no ill effects. Memory is an example of a preemptable resource. On the other hand, a nonpreemptable resource is one that cannot be taken away from process (without causing ill effect). For example, CD resources are not preemptable at an arbitrary moment.

Reallocating resources can resolve deadlocks that involve preemptable resources. Deadlocks that involve nonpreemptable resources are difficult to deal with.

3.2.3 Necessary and Sufficient Deadlock Conditions

Coffman (1971) identified four (4) conditions that must hold simultaneously for there to be a deadlock.

1. Mutual Exclusion Condition

The resources involved are non-shareable.

Explanation: At least one resource must be held in a non-shareable mode, that is, only one process at a time claims exclusive control of the resource. If another process requests that resource, the requesting process must be delayed until the resource has been released.

2. Hold and Wait Condition

Requesting process hold already, resources while waiting for requested resources.

Explanation: There must exist a process that is holding a resource already allocated to it while waiting for additional resource that are currently being held by other processes.

3. No-Preemptive Condition

Resources already allocated to a process cannot be preempted.

Explanation: Resources cannot be removed from the processes are used to completion or released voluntarily by the process holding it.

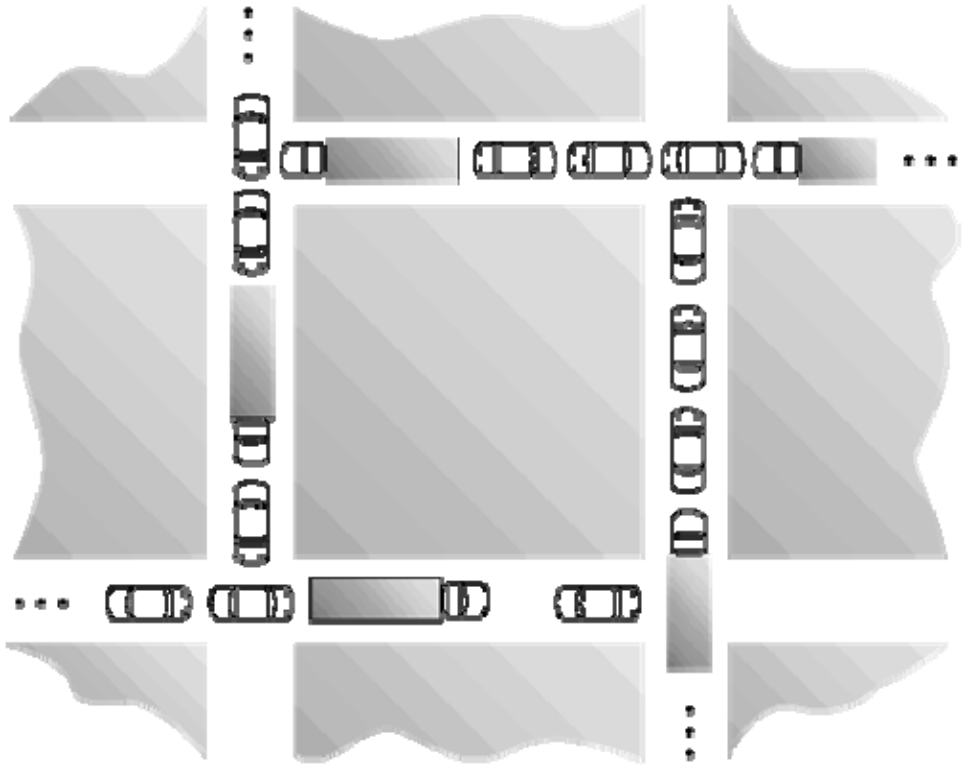
3. Circular Wait Condition

The processes in the system form a circular list or chain where each process in the list is waiting for a resource held by the next process in the list.

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

Conditions 1 and 3 pertain to resource utilization policies, while condition 2 pertains to resource requirements of individual processes. Only condition 4 pertains to relationships between resource requirements of a group of processes.

As an example, consider the traffic deadlock in the following figure:



Consider each section of the street as a resource.

1. Mutual exclusion condition applies, since only one vehicle can be on a section of the street at a time.
2. Hold-and-wait condition applies, since each vehicle is occupying a section of the street, and waiting to move on to the next section of the street.
3. No-preemptive condition applies, since a section of the street that is occupied by a vehicle cannot be taken away from it.
4. Circular wait condition applies, since each vehicle is waiting on the next vehicle to move. That is, each vehicle in the traffic is waiting for a section of street held by the next vehicle in the traffic.

The simple rule to avoid traffic deadlock is that a vehicle should only enter an intersection if it is assured that it will not have to stop inside the intersection.

It is not possible to have a deadlock involving only one single process. The deadlock involves a circular “hold-and-wait” condition between two or more processes, so “one” process cannot hold a resource, yet be waiting for another

resource that it is holding. In addition, deadlock is not possible between two threads in a process, because it is the process that holds resources, not the thread that is, each thread has access to the resources held by the process.

3.2.4 Resource-Allocation Graph

The deadlock conditions can be modeled using a directed graph called a resource allocation graph (RAG). A resource allocation graph is a directed graph. It consists of 2 kinds of nodes:

Boxes — Boxes represent resources, and Instances of the resource are represented as dots within the box i.e. how many units of that resource exist in the system.

Circles — Circles represent threads / processes. They may be a user process or a system process.

An edge can exist only between a process node and a resource node. There are 2 kinds of (directed) edges:

Request edge: It represents resource request. It starts from process and terminates to a resource. It indicates the process has requested the resource, and is waiting to acquire it.

Assignment edge: It represents resource allocation. It starts from resource instance and terminates to process. It indicates the process is holding the resource instance.

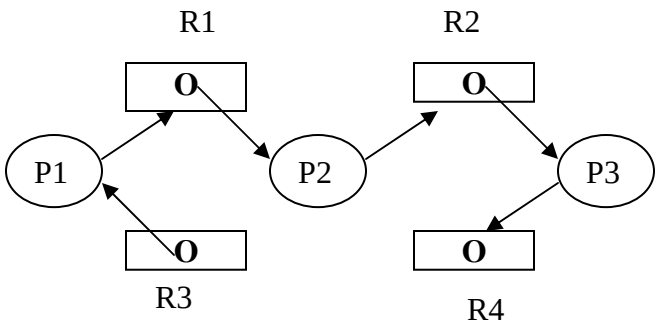
When a request is made, a request edge is added.

When request is fulfilled, the request edge is transformed into an assignment edge.

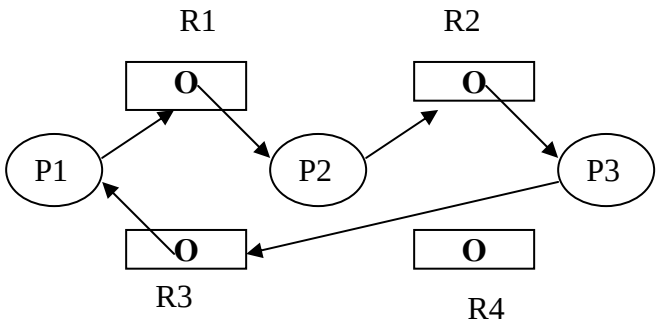
When process releases the resource, the assignment edge is deleted.

3.2.4.1 Interpreting a Resource Allocation Graph with Single Resource Instances

Following figure shows a resource allocation graph. If the graph does not contain a cycle, then no deadlock exists. Following figure is an example of a no deadlock situation.



If the graph does contain a cycle, then a deadlock does exist. As following resource allocation graph depicts a deadlock situation.



With single resource instances, a cycle is a necessary and sufficient condition for deadlock

So basic fact is that If graph contains no cycles then there is no deadlock. But If graph contains a cycle then there are two possibilities:

(a) If only one instance per resource type, then there is a deadlock.

(b) If several instances per resource type, possibility of deadlock is there.

3.2.5 Dealing with Deadlock

There are following approaches to deal with the problem of deadlock.

The Ostrich Approach: sticks your head in the sand and ignores the problem. This approach can be quite useful if you believe that they are rarest chances of deadlock occurrence. In that situation it is not a justifiable proposition to invest a lot in identifying deadlocks and tackling with it. Rather a better option is ignore it. For example if each PC deadlocks once per 100 years, the one reboot may be less painful that the restrictions needed to prevent it. But clearly it is not a good philosophy for nuclear missile launchers.

Deadlock prevention: This approach prevents deadlock from occurring by eliminating one of the four (4) deadlock conditions.

Deadlock detection algorithms: This approach detects when deadlock has occurred.

Deadlock recovery algorithms: After detecting the deadlock, it breaks the deadlock.

Deadlock avoidance algorithms: This approach considers resources currently available, resources allocated to each thread, and possible future requests, and only fulfill requests that will not lead to deadlock

3.2.6 Deadlock Prevention

Deadlock prevention is based on designing resource allocation policies, which make deadlocks impossible. Use of the deadlock prevention approach avoids the over- head of deadlock detection and resolution. However, it incurs two kinds of costs - overhead of using the resource allocation policy, and cost of resource idling due to the policy.

As described in earlier section, four conditions must hold for a resource deadlock to arise in a system:

- Non-shareable resources
- Hold-and-wait by processes
- No preemption of resources
- Circular waits.

Havender in his pioneering work showed that since all four of the conditions are necessary for deadlock to occur, it follows that deadlock might be prevented by denying any one of the conditions. Ensuring that one of these conditions cannot be satisfied prevents deadlocks. We first discuss how each of these conditions can be prevented and then discuss a couple of resource allocation policies based on the prevention approach.

3.2.6.1 Elimination of “Mutual Exclusion” Condition

The mutual exclusion condition must hold for non-sharable resources. That is, several processes cannot simultaneously share a single resource. This condition is difficult to eliminate because some resources, such as the tap drive and

printer, are inherently non-shareable. Note that shareable resources like read-only-file do not require mutually exclusive access and thus cannot be involved in deadlock.

Some resources can be made sharable. Spooling can make devices like printers or tape drives sharable. Spooling is storing the output on a shared medium, like disk, and using a single process to coordinate access to the shared resource. Print spooling is the cannonical example. There are generally some resources that cannot be spooled - semaphores, for example. Removing mutual exclusion is not always an option.

3.2.6.2 Elimination of "Hold and Wait" Condition

There are two possibilities for elimination of the second condition. The first alternative is that a process request be granted all of the resources it needs at once, prior to execution. The second alternative is to disallow a process from requesting resources whenever it has previously allocated resources. This strategy requires that all of the resources a process will need must be requested at once. The system must grant resources on "all or none" basis. If the complete set of resources needed by a process is not currently available, then the process must wait until the complete set is available. While the process waits, however, it may not hold any resources. Thus the "wait for" condition is denied and deadlocks simply cannot occur. This strategy can lead to serious waste of resources. For example, a program requiring ten tap drives must request and receive all ten derives before it begins executing. If the program needs only one tap drive to begin execution and then does not need the remaining tap drives for several hours. Then substantial computer resources (9 tape drives) will sit idle for several hours. This strategy can cause indefinite postponement (starvation). Since not all the required resources may become available at once.

3.2.6.3 Elimination of "No-preemption" Condition

The nonpreemption condition can be alleviated by forcing a process waiting for a resource that cannot immediately be allocated to relinquish all of its currently held resources, so that other processes may use them to finish. Suppose a system does allow processes to hold resources while requesting additional

resources. Consider what happens when a request cannot be satisfied. A process holds resources a second process may need in order to proceed while second process may hold the resources needed by the first process. This is a deadlock. This strategy requires that when a process that is holding some resources is denied a request for additional resources. The process must release its held resources and, if necessary, request them again together with additional resources. Implementation of this strategy denies the “no-preemptive” condition effectively.

The main drawback of this approach is high cost. When a process releases resources the process may lose all its work to that point. One serious consequence of this strategy is the possibility of indefinite postponement (starvation). A process might be held off indefinitely as it repeatedly requests and releases the same resources.

3.2.6.4 Elimination of “Circular Wait” Condition

Presence of a cycle in resource allocation graph indicates the “circular wait” condition. The last condition, the circular wait, can be denied by imposing a total ordering on all of the resource types and than forcing, all processes to request the resources in numerical order (increasing or decreasing). With this rule, the resource allocation graph can never have a cycle.

For example, provide a global numbering of all the resources, as shown

1 Card Reader
2 Printer
3 Plotter
4 Tape Drive
5 Card Punch

Now the rule is this: processes can request resources whenever they want to, but all requests must be made in numerical order. A process may request first printer and then a tape drive (order: 2, 4), but it may not request first a plotter and then a printer (order: 3, 2). The problem with this strategy is that it may be impossible to find an ordering that satisfies everyone. The resource ranking policy works best when all processes require their resources in the order of increasing ranks.

However, difficulty arises when a process requires resources in some other order. Now processes may tend to circumvent such difficulties by acquiring lower ranking resources much before they are actually needed. In the worst case this policy may degenerate into the 'all requests together' policy of resource allocation. Anyway this policy is attractive due to its simplicity once resource ranks have been assigned.

"All requests together" is the simplest of all deadlock prevention policies. A process must make its resource requests together-typically, at the start of its execution. This restriction permits a process to make only one multiple request in its lifetime. Since resources requested in a multiple request are allocated together, a blocked process does not hold any resources. The hold-and-wait condition is satisfied. Hence paths of length larger than 1 cannot exist in the Resource Allocation Graph, a mutual wait-for relationships cannot develop in the system. Thus, deadlocks cannot arise.

Example: The most common method of preventing deadlock is to prevent the circular wait. A simple way to do this, when possible, is to order the resources and always acquire them in order. Because a process can't be waiting on a lower numbered process while holding a higher numbered one, a cycle is impossible. One can consider the Dining Philosophers to be a deadlock problem, and can apply deadlock prevention to it by numbering the forks and always acquiring the lowest numbered fork first.

```
#define N 5 /* Number of philosphers */
#define RIGHT(i) (((i)+1) %N)
#define LEFT(i) (((i)==N) ? 0 : (i)+1)
typedef enum { THINKING, HUNGRY, EATING } phil_state;
phil_state state[N];
semaphore mutex =1;
semaphore f[N]; /* one per fork, all 1*/
void get_forks (int i)
{
        int max, min;
```

```
        if ( RIGHT(i) > LEFT(i) )
        {
                max = RIGHT(i); min = LEFT(i);
        }
        else
        {
                min = RIGHT(i); max = LEFT(i);
        }
        P (f [min]);
        P (f [max]);
}
void put_forks (int i)
{
        V (f [LEFT (i)]);
        V (f [RIGHT (i)]);
}
void philosopher (int process)
{
While (1)
        {
        think ();
        get_forks (process);
        eat ();
        put_forks (process);
        }
}
```

This solution doesn't get maximum parallelism, but it is an otherwise valid solution.

3.2.7 Deadlock Avoidance

This approach to the deadlock problem anticipates deadlock before it actually occurs. This approach employs an algorithm to access the possibility that

deadlock could occur and acting accordingly. This method differs from deadlock prevention, which guarantees that deadlock cannot occur by denying one of the necessary conditions of deadlock.

If the necessary conditions for a deadlock are in place, it is still possible to avoid deadlock by being careful when resources are allocated. Perhaps the most famous deadlock avoidance algorithm, due to Dijkstra [1965], is the Banker's algorithm. So named because the process is analogous to that used by a banker in deciding if a loan can be safely made.

3.2.7.1 Banker's Algorithm

In this analogy

Customers ≡ processes

Units ≡ resources, say, tape drive

Banker ≡ Operating System

Customers	Used	Max	
A	0	6	
B	0	5	Available
C	0	4	Units = 10
D	0	7	

In the above figure, we see four customers each of whom has been granted a number of credit units. The banker reserved only 10 units rather than 22 units to service them. At certain moment, the situation becomes

Customers	Used	Max	
A	1	6	
B	1	5	Available
C	2	4	Units = 2
D	4	7	

Safe State The key to a state being safe is that there is at least one way for all users to finish. In other analogy, the state of figure 2 is safe because with 2 units left, the banker can delay any request except C's, thus letting C finish and

release all four resources. With four units in hand, the banker can let either D or B have the necessary units and so on.

Unsafe State Consider what would happen if a request from B for one more unit were granted in above figure 2.

We would have following situation

Customers	Used	Max	
A	1	6	
B	2	5	Available
C	2	4	Units = 1
D	4	7	

This is an unsafe state.

If all the customers namely A, B, C, and D asked for their maximum loans, then banker could not satisfy any of them and we would have a deadlock.

Important Note: It is important to note that an unsafe state does not imply the existence or even the eventual existence a deadlock. What an unsafe state does imply is simply that some unfortunate sequence of events might lead to a deadlock.

The Banker's algorithm is thus to consider each request as it occurs, and see if granting it leads to a safe state. If it does, the request is granted, otherwise, it postponed until later. Haberman [1969] has shown that executing of the algorithm has complexity proportional to N^2 where N is the number of processes and since the algorithm is executed each time a resource request occurs, the overhead is significant.

3.2.7.2 Evaluation of Deadlock Avoidance Using the Banker's Algorithm

There are following advantages and disadvantages of deadlock avoidance using Banker's algorithm.

Advantages:

- There is no need to preempt resources and rollback state (as in deadlock detection and recovery)
- It is less restrictive than deadlock prevention

Disadvantages:

- In this case maximum resource requirement for each process must be stated in advance.
- Processes being considered must be independent (i.e., unconstrained by synchronization requirements)
- There must be a fixed number of resources (i.e., can't add resources, resources can't break) and processes (i.e., can't add or delete processes)
- Huge overhead — Operating system must use the algorithm every time a resource is requested. So a huge overhead is involved.

3.2.8 Deadlock Detection

Deadlock detection is the process of actually determining that a deadlock exists and identifying the processes and resources involved in the deadlock. The basic idea is to check allocation against resource availability for all possible allocation sequences to determine if the system is in deadlocked state. Of course, the deadlock detection algorithm is only half of this strategy. Once a deadlock is detected, there needs to be a way to recover. Several alternatives exist:

- Temporarily prevent resources from deadlocked processes.
- Back off a process to some check point allowing preemption of a needed resource and restarting the process at the checkpoint later.
- Successively kill processes until the system is deadlock free.

These methods are expensive in the sense that each iteration calls the detection algorithm until the system proves to be deadlock free. The complexity of algorithm is O (N^2) where N is the number of processes. Another potential problem is starvation; same process killed repeatedly.

3.2.9 Deadlock Recovery

Once you have discovered that there is a deadlock, what do you do about it? One thing to do is simply re-boot. A less drastic approach is to yank back a resource from a process to break a cycle. As we saw, if there are no cycles, there is no deadlock. If the resource is not preemptable, snatching it back from a process may do irreparable harm to the process. It may be necessary to kill the

process, under the principle that at least that's better than crashing the whole system.

So once a deadlock has been detected, one of the 4 conditions must be invalidated to remove the deadlock. Generally, the system acts to remove the circular wait, because making a system suddenly preemptive with respect to resources, or making a resource suddenly sharable is usually impractical. Because resources are generally not dynamic, the easiest way to break such a cycle is to terminate a process.

Usually the process is chosen at random, but if more is known about the processes, that information can be used. For example the largest or smallest process can be disabled or the one waiting the longest.

Such discrimination is based on assumptions about the system workload.

Some systems facilitate deadlock recovery by implementing check pointing and rollback. Check pointing is saving enough state of a process so that the process can be restarted at the point in the computation where the checkpoint was taken. Auto saving file edits is a form of check pointing. Check pointing costs depend on the underlying algorithm. Very simple algorithms (like linear primality testing) can be checkpointed with a few words of data. More complicated processes may have to save all the process state and memory.

Checkpoints are taken less frequently than deadlock is checked for. If a deadlock is detected, one or more processes are restarted from their last checkpoint. The process of restarting a process from a checkpoint is called rollback. The hope is that the resource requests will not interleave again to produce deadlock.

Deadlock recovery is generally used when deadlocks are rare, and the cost of recovery (process termination or rollback) is low. Process check pointing can also be used to improve reliability (long running computations), assist in process migration (Sprite, Mach), or reduce startup costs (emacs).

Database systems use checkpoints, as well as a technique called logging, allowing them to run processes “backwards,” undoing everything they have done. It works like this: Each time the process performs an action, it writes a log record containing enough information to undo the action. For example, if the action is to

assign a value to a variable, the log record contains the previous value of the record. When a database discovers a deadlock, it picks a victim and rolls it back.

Rolling back processes involved in deadlocks can lead to a form of starvation, if we always choose the same victim. We can avoid this problem by always choosing the youngest process in a cycle. After being rolled back enough times, a process will grow old enough that it never gets chosen as the victim--at worst by the time it is the oldest process in the system. If deadlock recovery involves killing a process altogether and restarting it, it is important to mark the “starting time” of the reincarnated process as being that of its original version, so that it will look older that new processes started since then.

When should you check for deadlock? There is no one best answers to this question; it depends on the situation. The most “eager” approach is to check whenever we do something that might create a deadlock. Since a process cannot create a deadlock when releasing resources, we only have to check on allocation requests. If the Operating System always grants requests as soon as possible, a successful request also cannot create a deadlock. Thus we only have to check for a deadlock when a process becomes blocked because it made a request that cannot be immediately granted. However, even that may be too frequent. As we saw, the deadlock-detection algorithm can be quite expensive if there are a lot of processes and resources, and if deadlock is rare, we can waste a lot of time checking for deadlock every time a request has to be blocked.

What's the cost of delaying detection of deadlock? One possible cost is poor CPU utilization. In an extreme case, if all processes are involved in a deadlock, the CPU will be completely idle. Even if there are some processes that are not deadlocked, they may all be blocked for other reasons (e.g. waiting for I/O). Thus if CPU utilization drops, that might be a sign that it's time to check for deadlock. Besides, if the CPU isn't being used for other things, you might as well use it to check for deadlock!

On the other hand, there might be a deadlock, but enough non-deadlocked processes to keep the system busy. Things look fine from the point of view of the Operating System, but from the selfish point of view of the deadlocked

processes, things are definitely not fine. If the processes may represent interactive users, who can't understand why they are getting no response. Worse still, they may represent time-critical processes (missile defense, factory control, hospital intensive care monitoring, etc.) where something disastrous can happen if the deadlock is not detected and corrected quickly. Thus another reason to check for deadlock is that a process has been blocked on a resource request "too long." The definition of "too long" can vary widely from process to process. It depends both on how long the process can reasonably expect to wait for the request, and how urgent the response is. If an overnight run deadlocks at 11pm and nobody is going to look at its output until 9am the next day, it doesn't matter whether the deadlock is detected at 11:01pm or 8:59am. If all the processes in a system are sufficiently similar, it may be adequate simply to check for deadlock at periodic intervals (e.g., one every 5 minutes in a batch system; once every millisecond in a real-time control system).

3.2.10 Mixed approaches to deadlock handling

The deadlock handling approaches differ in terms of their usage implications. Hence it is not possible to use a single deadlock handling approach to govern the allocation of all resources. The following mixed approach is found useful:

1. **System control block:** Control blocks like JCB, PCB etc. can be acquired in a specific order. Hence resource ranking can be used here. If a simpler strategy is desired, all control blocks for a job or process can be allocated together at its initiation.
2. **I/O devices files:** Avoidance is the only practical strategy for these resources. However, in order to eliminate the overheads of avoidance, new devices are added as and when needed. This is done using the concept of spooling. If a system has only one printer, many printers are created by using some disk area to store a file to be printed. Actual printing takes place when a printer becomes available.
3. **Main memory:** No deadlock handling is explicitly necessary. The memory allocated to a program is simply preempted by swapping out the program whenever the memory is needed for another program.

3.2.11 Evaluating the Approaches to Dealing with Deadlock

- The Ostrich Approach — ignoring the problem
  It is a good solution if deadlock is not frequent.
- Deadlock prevention — eliminating one of the four (4) deadlock conditions
  This approach may be overly restrictive and results into the under utilization of the resources.
- Deadlock detection and recovery — detect when deadlock has occurred, then break the deadlock
  In it there is a tradeoff between frequency of detection and performance / overhead added.
- Deadlock avoidance — only fulfilling requests that will not lead to deadlock
  It needs too much a priori information and not very dynamic (can't add processes or resources), and involves huge overhead

3.3 Summary

- A set of process is in a deadlock state if each process in the set is waiting for an event that can be caused by only another process in the set. Processes compete for physical and logical resources in the system. Deadlock affects the progress of processes by causing indefinite delays in resource allocation.
- *There are four Necessary and Sufficient Deadlock Conditions (1) Mutual Exclusion Condition: The resources involved are non-shareable, (2) Hold and Wait Condition: Requesting process hold already, resources while waiting for requested resources,(3) No-Preemptive Condition: Resources already allocated to a process cannot be preempted,(4) Circular Wait Condition: The processes in the system form a circular list or chain where each process in the list is waiting for a resource held by the next process in the list.*
- The deadlock conditions can be modeled using a directed graph called a resource allocation graph (RAG) consisting of boxes (resource), circles (process) and edges (request edge and assignment edge). The resource allocation graph helps in identifying the deadlocks.
- There are following approaches to deal with the problem of deadlock: (1) The Ostrich Approach — stick your head in the sand and ignore the problem, (2)

Deadlock prevention — prevent deadlock from occurring by eliminating one of the 4 deadlock conditions, (3) Deadlock detection algorithms — detect when deadlock has occurred, (4) Deadlock recovery algorithms — break the deadlock, (5) Deadlock avoidance algorithms — consider resources currently available, resources allocated to each thread, and possible future requests, and only fulfill requests that will not lead to deadlock

- There are merits/demerits of each approach. The Ostrich Approach is a good solution if deadlock is not frequent. Deadlock prevention may be overly restrictive. In Deadlock detection and recovery there is a tradeoff between frequency of detection and performance / overhead added, Deadlock avoidance needs too much a priori information and not very dynamic (can't add processes or resources), and involves huge overhead

3.4 Keywords

Deadlock: A deadlock is a situation in which some processes in the system face indefinite delays in resource allocation.

Preemptable resource: A preemptable resource is one that can be taken away from the process with no ill effects.

Nonpreemptable resource: It is one that cannot be taken away from process (without causing ill effect).

Mutual exclusion: several processes cannot simultaneously share a single resource

3.5 SELF-ASSESMENT QUESTIONS (SAQ)

1. What do you understand by deadlock? What are the necessary conditions for deadlock?
2. What do you understand by resource allocation graph (RAG)? Explain using suitable examples, how can you use it to detect the deadlock?
3. What do you mean by pre-emption and non-preemption discuss with an example?
4. Compare and contrast the following policies of resource allocation:

(a) All resources requests together.

(b) Allocation using resource ranking.

(c) Allocation using Banker's algorithm

On the basis of (a) resource idling and (b) overhead of the resource allocation algorithm.

5. How can pre-emption be used to resolve deadlock?
6. Why Banker's algorithm is called so?
7. Under what condition(s) a wait state becomes a deadlock?
8. Explain how mutual exclusion prevents deadlock.
9. Discuss the merits and demerits of each approach dealing with the problem of deadlock.
10. Differentiate between deadlock avoidance and deadlock prevention.
11. A system contains 6 units of a resource, and 3 processes that need to use this resource. If the maximum resource requirement of each process is 3 units, will the system be free of deadlocks for all time? Explain clearly.

    If the system had 7 units of the resource, would the system be deadlock-free?

3.6 SUGGESTED READINGS / REFERENCE MATERIAL

1. Operating System Concepts, 5th Edition, Silberschatz A., Galvin P.B., John Wiley and Sons.
2. Systems Programming and Operating Systems, 2nd Revised Edition, Dhamdhere D.M., Tata McGraw Hill Publishing Company Ltd., New Delhi.
3. Operating Systems, Madnick S.E., Donovan J.T., Tata McGraw Hill Publishing Company Ltd., New Delhi.
4. Operating Systems-A Modern Perspective, Gary Nutt, Pearson Education Asia, 2000.
5. Operating Systems, Harris J.A., Tata McGraw Hill Publishing Company Ltd., New Delhi, 2002.
6. Operating Systems, A Concept-based Approach, Dhamdhere D.M., Tata McGraw Hill Publishing Company Ltd., New Delhi.

Lesson number: 4	***Writer: Dr. Rakesh Kumar***
Memory Management	***Vetter: Dr. Pradeep Bhatia***

4.0 Objectives

Memory is a very important resource of the computer system to be managed by the operating system. The objectives of this lesson are to get the students familiar with the various concepts of memory management in particular:

(a) Segmentation

(b) Paging

(c) Virtual memory

4.1 Introduction

Memory management module is concerned with (a) Keeping track of whether each location is allocated or unallocated, to which process and how much, (b) If memory is to be shared by more than one process concurrently, it must be determined which process' request should be satisfied, (c) Once it is decided to allocate memory, the specific locations must be selected and allocated. Memory status information is updated, and (d) Handling the deallocation/reclamation of memory. After the process holding memory is finished, memory locations held by it are declared free by changing the status information. In order to accomplish these, there are varieties of memory management systems. They are:

1. Contiguous, real memory management system such as:
   - Single, contiguous memory management system
   - Fixed partitioned memory management system
   - Variable Partitioned memory management system
2. Non-Contiguous, real memory management system
   - Paged memory management system
   - Segmented memory management system
   - Combined memory management system
3. Non-Contiguous, virtual memory management system
   - Virtual memory management system

4.2 Presentation of Contents

4.2.1 Contiguous Memory Management

4.2.1.1 Single Contiguous Memory Management

4.2.1.2 Fixed Partitioned Memory Management System

4.2.1.3 Variable Partitioned Memory Allocation

4.2.2 Noncontiguous memory management

4.2.2.1 Segmentation

4.2.2.1.1 Address Translation

4.2.2.1.2 Segment Descriptor Caching

4.2.2.1.3 Protection

4.2.2.1.4 Sharing

4.2.2.2 Paging

4.2.2.2.1 Page Allocation

4.2.2.2.2 Hardware Support for Paging

4.2.2.2.3 Protection and Sharing

4.2.2.3 Virtual Memory

4.2.2.3.1 Principles of Operation

4.2.2.3.2 Management of Virtual Memory

4.2.2.3.3 Program Behavior

4.2.2.3.4 Replacement Policies

4.2.2.3.5 Replacement Algorithms

4.2.2.3.6 Allocation Policies

4.2.2.3.7 Hardware Support and Considerations

4.2.2.3.8 Protection and Sharing

4.2.2.4 Segmentation and Paging

4.2.1 Contiguous Memory Management:

In Contiguous Memory Management each program occupies a single contiguous block of storage locations.

4.2.1.1 Single Contiguous Memory Management

In this scheme, the physical memory is divided into two contiguous areas. One of them is permanently allocated to the resident portion of the OS. The remaining memory is allocated to user processes, which are loaded and executed one at a time, in response to user commands. This process is run to completion and then the next process is brought in memory.

4.2.1.2 Fixed Partitioned Memory Management System

In this scheme, memory is divided into number of contiguous regions called partitions, could be of different sizes. But once decided, they could not be changed. Partitions are fixed at the time of system generation, a process of setting the OS to specific requirements. There are two forms of memory partitioning (i) Fixed Partitioning and (ii) Variable Partitioning.

In fixed partitioning the main memory is divided into fixed number of partitions during system startup. The number and sizes of individual partitions are decided by the factors like capacity of the available physical memory, desired degree of multiprogramming, and the typical sizes of processes most frequently run on a given installation. The number of partitions represents an upper limit on degree of multiprogramming. On request for partitions due to (1) creations of new processes or (2) reactivations of swapped-out processes, the memory manager attempts to satisfy these requests from the pool of free partitions. Common obstacles faced by it are: (1) All partitions are allocated, or (2) No free partition is large enough to accommodate the incoming process i.e. fragmentation. It refers to the unused memory that the memory management system cannot allocate. It is of two types: External and Internal. External Fragmentation is waste of memory between partitions caused by scattered non-contiguous free space. It occurs when total available memory space is enough to satisfy the request for a process to be allocated, but it is not continuous. It is severe in variable size partitioning schemes. Internal fragmentation is waste of memory within a partition caused by difference between size of partition and the process allocated. It refers to the amount of memory, which is not being used and is allocated along with a process request. It is severe in fixed partitioning schemes.

The main problem with fixed partitioned memory management system is determining the best region size to minimize the problem of fragmentation. It is difficult to achieve in fixed partitioning because in it the number of partitions and their sizes are decided statically and with a dynamic set of job to run there is no one right partition of memory.

4.2.1.3 Variable Partitioned Memory Allocation

In variable partitions, the number of partitions and their sizes are variable as they are not defined at the time of system generation. Starting with the initial state of the system, partitions are created dynamically to fit the needs of each requesting process. When a process departs, the memory manager returns the vacated space to the pool of free memory areas from which partition allocations are

made. The OS obviously needs to keep track of both partitions and free memory. Once created, a partition is defined by its base address and size. Free areas of memory are produced upon termination of partitions and as leftovers in the partition creation process. For allocation and for partition creation purpose, the OS must keep track of the starting address and size of each free area of memory. The highly dynamic nature of both the number and the attributes of free areas suggest the use of some sort of a linked list to describe them within the free memory itself. Common algorithms for selection of a free area of memory are (a) First fit, (b) Best fit, (c) Worst fit, and (d) Next Fit.

First fit is faster because it terminates as soon as a free block large enough to house a new partition is found but it does not minimize wasted memory for a given allocation. Best fit searches the entire free list to find the smallest free block large enough to hold a partition being created. Best fit is slower, and it tends to produce small leftover free blocks that may be too small for subsequent allocations. Neither algorithm has been shown to be superior to the other in terms of wasted memory. Next fit is a modification of first fit whereby the pointer to the free list is saved following an allocation and used to begin the search for the subsequent allocation as opposed to always starting from the beginning of the free list. The idea is to reduce the search by avoiding examination of smaller blocks that tend to be created at the beginning of the free list as a result of previous allocations. Worst fit allocates the largest free block to reduce the rate of production of small holes. Simulation studies indicate that worst fit allocation is not very effective in reducing wasted memory in the processing of a series of requests.

4.2.2 Noncontiguous memory management

In Non-Contiguous Memory Management a program is divided into several blocks that may be placed throughout main storage in pieces not necessarily adjacent to one another. It is done in various ways broadly categorized as (a)

Non-Contiguous, real memory management system & (b) Non-Contiguous, virtual memory management system.

4.2.2.1 Segmentation

Segments are formed at program translation time by grouping together logically related items. Programs are collections of subroutines, stacks, functions etc. Each of these components is of variable length and are logically related entities. All segments of all programs do not have to be of the same length. There is a maximum segment length. Although different segments may be placed in separate, noncontiguous areas of physical memory, items belonging to a single segment must be placed in contiguous areas of physical memory.

Segmentation is mapping of user's view onto physical memory. A logical address space is a collection of segments. Each segment has a name and length. User specifies each address by segment name or number and offset within segment. Segments are numbered and are referenced by segment number. For relocation purposes, each segment is compiled to begin at its own virtual address 0. An individual item within a segment is then identifiable by its offset relative to the beginning of the enclosing segment. Thus, logical address consists of <segment no., offset>. To simplify processing, segment names are usually mapped to (virtual) segment numbers. This mapping is static, and systems programs in the course of preparation of process images may perform it.

4.2.2.1.1 Address Translation

Since physical memory in segmented systems generally retains its linear-array organization, some address translation mechanism is needed to convert a two-dimensional virtual-segment address into its one-dimensional physical equivalent. In segmented systems, items belonging to a single segment reside in one contiguous area of physical memory. With each segment compiled as if starting from the virtual address zero, segments are generally individually relocatable. As a result, different segments of the same process need not occupy contiguous areas of physical memory.

When requested to load a segmented process, the OS attempts to allocate memory for the supplied segments. Using logic similar to that used for dynamic partitioning, it may create a separate partition to suit the needs of each particular segment. The base (obtained during partition creation) and size (specified in the load module) of a loaded segment are recorded as a tuple called the segment descriptor. All segment descriptors of a given process are collected in a table called the segment descriptor table (SDT). Two dimensional user defined

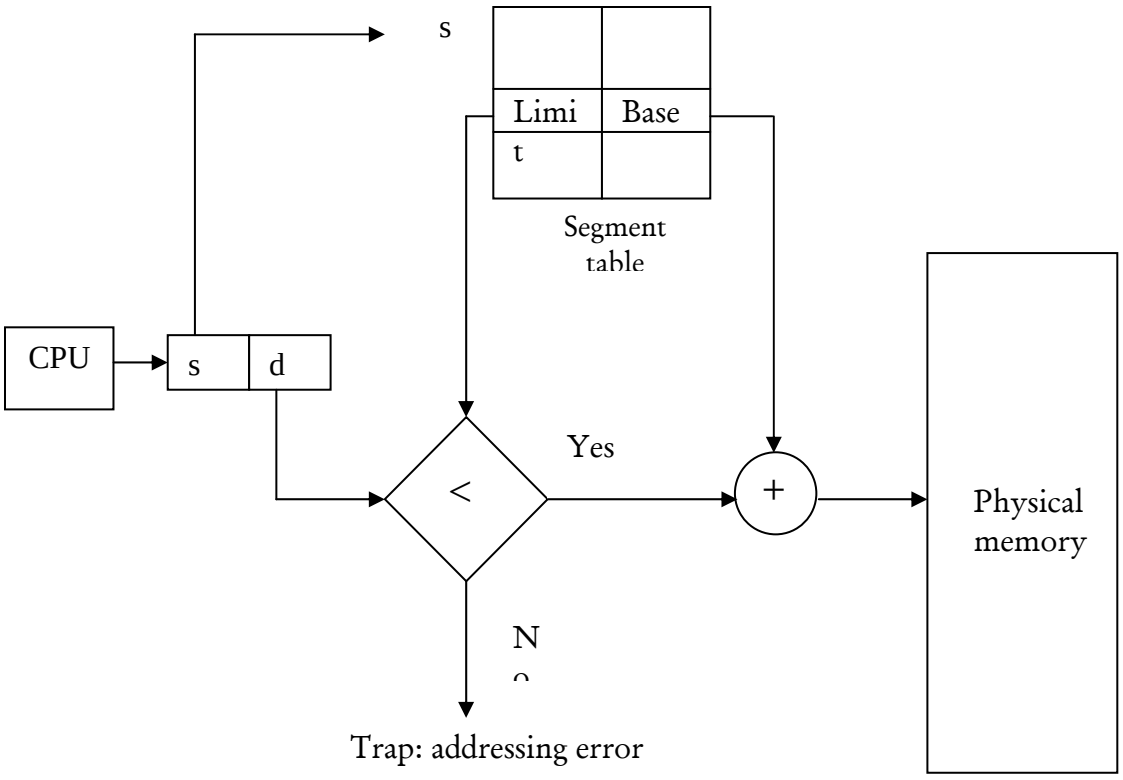
address is mapped to one dimensional physical address by segment descriptor table. Each entry of this table has segment base and segment limit. Segment base contains the starting physical address of the segment and segment limit specifies the length of the segment.

Figure 1 illustrates a sample placement of the segments into physical memory, and the resulting SDT formed by the Operating System. With the physical base address of each segment defined, the process of translation of a virtual, two-component address into its physical equivalent basically follows the mechanics of based addressing. The segment number provided in the virtual address is used to index the segment descriptor table and to obtain the physical base address of the related segment. Adding the offset of the desired item to the base of its enclosing segment then produces the physical address.

The size of a SDT is related to the size of the virtual address space of a process. Given their potential size, SDT are not kept in registers. Being a collection of logically related items, the SDTs themselves are often treated as special types of segments. Their accessing is usually facilitated by means of a dedicated hardware register called the segment descriptor table base register (SDTBR), which is set to point to the base of the running process's SDT. Since the size of an SDT may vary from a few entries to several thousand, another dedicated hardware register, called the segment descriptor table limit register (SDTLR), is provided to mark the end of the SDT pointed to by the SDTBR. In this way, an SDT need contain only as many entries as there are segments actually defined in a given process. Attempts to access nonexistent segments may be detected and dealt with as nonexistent-segment exceptions. Mapping each virtual address requires two physical memory references for a single virtual (program) reference, as follows:

- Memory reference to access the segment descriptor in the SDT
- Memory reference to access the target item in physical memory

In other words, segmentation may cut the effective memory bandwidth in half by making the effective virtual-access time twice as long as the physical memory access time.



Hardware support for Segmentation

Figure 1 – Address translation in segmented systems

4.2.2.1.2 Segment Descriptor Caching

As performance of segmented systems is dependent on the address translation process, system designers often provide some hardware accelerators to speed the translation. Memory references expended on mapping may be avoided by keeping segment descriptors in registers. However to keep an entire SDT of the running process in register is very costly, hence most frequently used segment descriptors are kept in registers. In this way, most of the memory references may be mapped with the aid of registers. The rest may be mapped using the SDT in memory, as usual. This scheme is dependent on the OS's ability to select the proper segment descriptors for storing into registers. In order to provide the intuitive motivation for one possible implementation of systematic descriptor selection, let us investigate the types of segments referenced by the executing process.

Memory references may be functionally categorized as accesses to (i) Instructions, (ii) Data, and (iii) Stack. A typical instruction execution sequence consists of a mixture of the outline types of memory references. In fact, completion of a single stack manipulation instruction, such as a push of a datum from memory onto stack, may require all three types of references. Thus the working space of a process normally encompasses one each of code, data, and stack segments. Therefore, keeping the current code, data, and stack segment descriptors in registers may accelerate address translation. Depending on its type, a particular memory reference may then be mapped using the appropriate register. But can we know the exact type of each memory reference as the processor is making it? The answer is yes, with the proper hardware support. Namely, in most segmented machines the CPU emits a few status bits to indicate the type of each memory reference. The memory management hardware uses this information to select the appropriate mapping register.

Register-assisted translation of virtual to physical addresses is illustrated in Figure 2. As indicated, the CPU status lines are used to select the appropriate segment descriptor register (SDR). The size field of the selected segment descriptor is used to check whether the intended reference is within the bounds of the target segment. If so, the base field is added with the offset to produce the physical address. By making the choice of the appropriate segment register implicit in the type of memory reference being made, segment typing may eliminate the need to keep track of segment numbers during address translations. Though segment typing is certainly useful, it may become restrictive at times. For example, copying an instruction sequence from one segment into another may confuse the selector logic into believing that source and target segments should be of type data rather than code. Using the so-called segment override of type prefixes, which allows the programmer to explicitly indicate the particular segment descriptor register to be used for mapping the memory reference in question, may alleviate this problem.

Base DS1	Size DS1
Base DS2	Size DS2
Base SS1	Size SS1

CS2

Segment Size

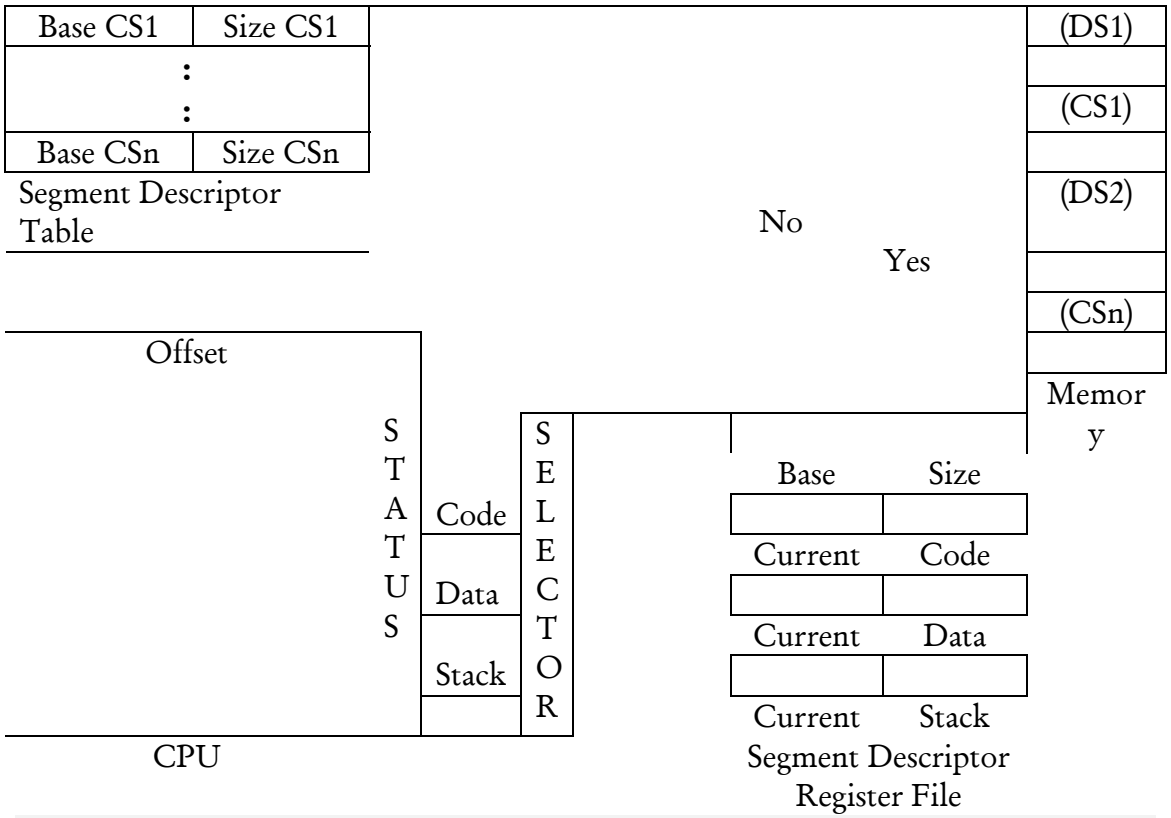


Figure 2 – Segment-descriptor cache registers

Segment descriptor registers are initially loaded from the SDT. Whenever the running process makes an intersegment reference, the corresponding segment descriptor is loaded into the appropriate register from the SDT. For example, an intersegment JUMP or CALL causes the segment descriptor of the target (code) segment to be copied from the SDT to the code segment descriptor register. When segment typing is used as described, segment descriptor caching becomes deterministic as opposed to probabilistic. Segment descriptors stored in the three segment descriptor registers; define the current working set of the executing process. Since membership in the working set of segments of a process changes with time, segment descriptor registers are normally included in the process state. Upon each process switch, the contents of the SDRs of the departing process are stored with the rest of its context. Before dispatching the new running process, the OS loads segment descriptor registers with their images recorded in the related PCB.

4.2.2.1.3 Protection

The base-limit form of protection is obviously the most natural choice for segmented systems. The legal address space of a process is the collection of segments defined by its SDT. Except for shared segments. Placing different segments in disjoint areas of memory enforces separation of distinct address space. An interesting possibility in segmented systems is to provide protection within the address space of a single process, in addition to the more usually protection between different processes. Given that the type of each segment is defined commensurate with the nature of information stored in its constituent elements, access rights to each segment can be defined accordingly. For instance, though both reading and writing of stack segments may be necessary, accessing of code segments can be permitted in execute-only or perhaps in the read-only mode. Data segments can be read-only, write-only, or read-write. Thus, segmented systems may be able to prohibit some meaningless operations. Additional examples include prevention of stack growth into the adjacent code or data areas, and other errors resulting from mismatching of segment types and intended references to them. An important observation is that access rights to different portions of a single address space may vary in accordance with the type of information stored therein. Due to the grouping of logically related items, segmentation is one of the rare memory-management schemes that allow such finely grained delineation of access rights. The mechanism for enforcement of declared access rights in segmented systems is usually coupled with the address translation hardware. Typically, access-rights bits are included in segment descriptors. In the course of address mapping, the intended type of reference is checked against the access rights for the segment in question. Any mismatch results in abortion of the memory reference in progress, and a trap to the OS.

4.2.2.1.4 Sharing

Shared objects are usually placed in separate, dedicated segments. A shared segment may be mapped, via the appropriate SDTs, to the virtual-address spaces of all processes that are authorized to reference it. The deliberate use of offsets and of bases addressing facilitate sharing since the virtual offset of a given item is identical in all processes that share it. The virtual number of a

shared segment, on the other hand, need not be identical in all address spaces of which it is a member. These points are illustrated in Figure 3, where a code segment EMACS is assumed to be shared by three processes P1, P2 & P3. The relevant portions of the SDTs of the participating processes P1, P2 & P3 are SDT1, SDT2, and SDT3 respectively, and shown. As indicated, the segment EMACS is assumed to have different virtual numbers in the three address spaces of which it is part. The placement of access-rights bits in segment descriptor tables is also shown. Figure 3 illustrates the fact that different processes can have different access rights to the same shared segment. For example, whereas processes P1 and P2 can execute only the shared segment EMACS, process P3 is allowed both reading and writing.

Figure 3 also illustrates the ability of segmented systems to conserve memory by sharing the code of programs executed by many users. In particular, each participating process can execute the shared code from EMACS using its own private data segment. Assuming there is an editor, this means that a single copy of it may serve the entire user population of a time-sharing system. Naturally, execution of EMACS on behalf of each user is stored in a private data segment of its corresponding process. For example, users 1, 2, and 3 can have their respective texts buffers stored in data segments DATA1, DATA2, and DATA3. Depending on which of the three processes is active at a given time, the hardware data segment descriptor register points to data segment DATA1, DATA2, or DATA3, and the code segment descriptor register points to EMACS in all cases. Of course, the current instruction to be executed by the particular process is indicated by the program counter, which is saved and restored as a part of each process's state. In segmented systems, the program counter register usually contains offsets of instructions within the current code segment. This facilitates sharing by making all code self-references relative to the beginning of the current code segment. When coupled with segment typing, this feature makes it possible to assign different virtual segment numbers to the same (physical) shared segment in virtual-address spaces of different processes of

which it is a part. Alternatively, the problem of making direct self-references in shared routines restricts the type of code that may safely be shared.

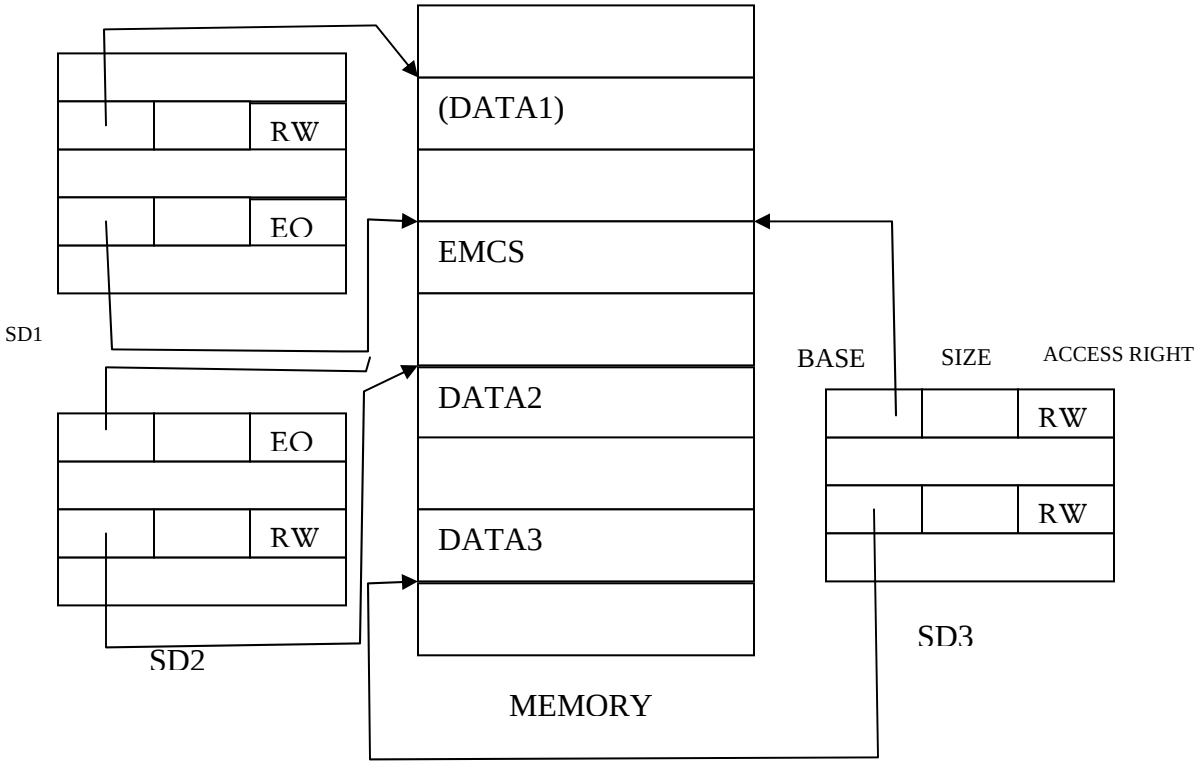


Figure 4 – Sharing in segmented systems

As described, sharing is encouraged in segmented systems. This presents some problems in systems that also support swapping, which is normally done to increase processor utilization. For example, a shared segment may need to maintain its memory residence while being actively used by any of the processes authorized to reference it. Swapping in this case opens up the possibility that a participating process may be swapped out while its shared segment remains resident. When such a process is swapped back in, the construction of its SDT must take into consideration the fact that the shared segment may already be resident. In other words, the OS must keep track of shared segments and of processes that access them. When a participating process is loaded in memory, the OS is expected to identify the location of the shared segment in memory, if

any, and to ensure its proper mapping from all virtual address spaces of which it is a part.

4.2.2.2 Paging

In it, the physical memory is conceptually divided into a number of fixed-size slots, called page frames. The virtual-address space of a process is also split into fixed-size blocks of the same size, called pages. Memory management module identifies sufficient number of unused page frames for loading of the requesting process's pages. An address translation mechanism is used to map virtual pages to their physical counterparts. Since each page is mapped separately, different page frames allocated to a single process need not occupy contiguous areas of physical memory.

Figure 4 demonstrates the basic principle of paging. It illustrates a sample 16 MB system where virtual and physical addresses are assumed to be 24 bits long each. The page size is assumed to be 4096 bytes. Thus, the physical memory can accommodate 4096 page frames of 4096 bytes each. After reserving 1 MB of physical memory for the resident portion of the Operating System, the remaining 3840 page frames are available for allocation to user processes. The addresses are given in hexadecimal notation. Each page is 1000H bytes long, and the first user-allocatable page frame starts at the physical address 100000H. The virtual-address space of a sample user process that is 14,848 bytes (3A00H) long is divided into four virtual pages numbered from 0 to 3. A possible placement of those pages into physical memory is depicted in Figure 4. The mapping of virtual addresses to physical addresses in paging systems is performed at the page level. Each virtual address is divided into two parts: the page number and the offset within that page. Since pages and page frames have identical sizes, offsets within each are identical and need not be mapped. So each 24-bit virtual address consists of a 12-bit page number (high-order bits) and a 12-bit offset within the page.

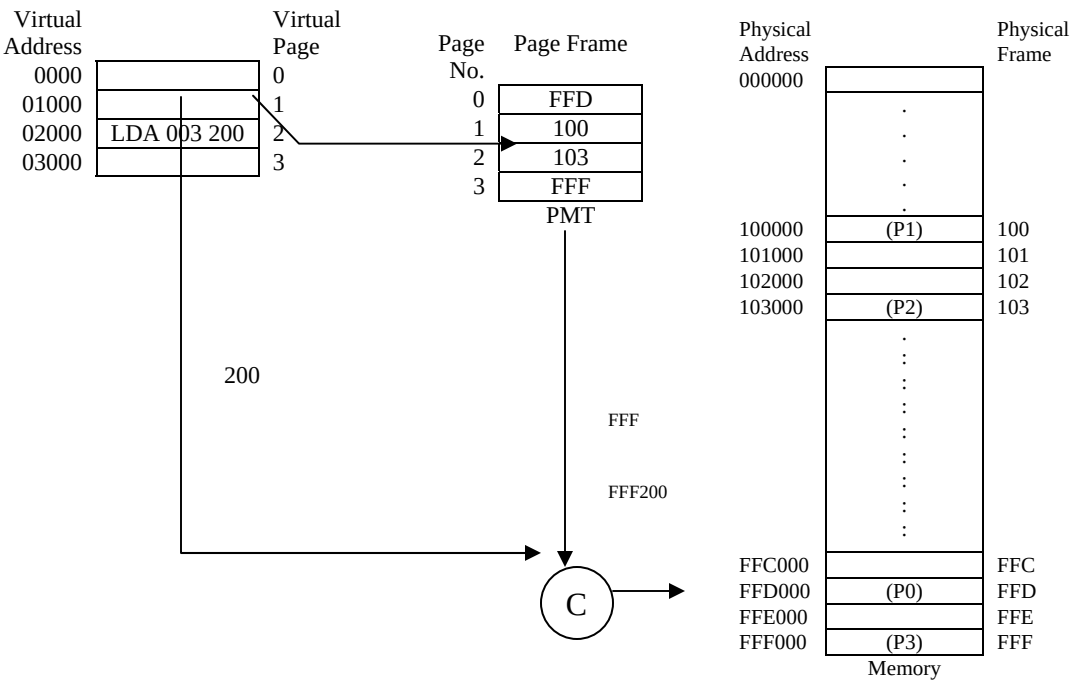


Figure 4 – Paging

Address translation is performed with the help of the page-map table (PMT), constructed at process-loading time. As indicated in figure 4, there is one PMT entry for each virtual page of a process. The value of each entry is the number of the page frame in the physical memory where the corresponding virtual page is placed. Since offsets are not mapped, only the page frame number need be stored in a PMT entry. E.g., virtual page 0 is assumed to be placed in the physical page frame whose starting address is FFD000H (16,764,928 decimal). With each frame being 1000H bytes long, the corresponding page frame number is FFDH, as indicated on the right-hand side of the physical memory layout in Figure 4. This value is stored in the first entry of the PMT. All other PMT entries are filled with page frame numbers of the region where the corresponding pages are actually loaded.

The logic of the address translation process in paged systems is illustrated in Figure 4 on the example of the virtual address 03200H. The virtual address is split by hardware into the page number 003H, and the offset within that page (200H). The page number is used to index the PMT and to obtain the corresponding physical frame number, i.e. FFF. This value is then concatenated with the offset to produce the physical address, FFF200H, which is used to reference the target item in memory.

The OS keeps track of the status of each page frame by using a memory-map table (MMT). Format of an MMT is illustrated in Figure 5, assuming that only the process depicted in Figure 5 and the OS are resident in memory.

000	ALLOCATED
	:
	:
	:
0FF	ALLOCATED
100	ALLOCATED
101	FREE
102	FREE
103	ALLOCATED
	:
	:
	:
FFC	FREE
FFD	ALLOCATED
FFE	FREE
FFF	ALLOCATED

Figure 5 – Memory-map table (MMT)

Each entry of the MMT described the status of page frame as FREE or ALLOCATED. The number of MMT entries i.e. ***f*** is computed as ***f = m/p*** where ***m*** is the size of the physical memory, and ***p*** is page size. Both ***m*** and ***p*** are usually an integer power of base 2, thus resulting in f being an integer. When requested to load a process of size ***s***, the OS must allocate n free page frames, so that ***n*** = Round(***s/p***) where ***p*** is the page size. The OS allocates memory in terms of an integral number of page frames. If the size of a given process is not a multiple of the page size, the last page frame may be partly unused resulting into page fragmentation.

After selecting n free page frames, the OS loads process pages into them and constructs the page-map table of the process. Thus, there is one MMT per system, and as many PMTs as there are active processes. When a process

terminates or becomes swapped out, memory is deallocated by releasing the frame holdings of the departing process to the pool of free page frames.

4.2.2.2.1 Page Allocation

The efficiency of the memory allocation algorithm depends on the speed with which it can locate free page frames. To facilitate this, a list of free pages is maintained instead of the static-table format of the memory map assumed earlier. In that case, n free frames may be identified and allocated by unlinking the first n nodes of the free list. Deallocation of memory in systems without the free list consists of marking in the MMT as FREE all frames found in the PMT of the departing process a time consuming operation. Frames identified in the PMT of the departing process can be linked to the beginning of the freed list. Linking at the beginning is the fastest way of adding entries to an unordered singly linked list. Since the time complexity of deallocation is not significantly affected by the choice of data structure of free pages, the free-list approach has a performance advantage as its time complexity of deallocation is not significantly affected by the choice of data structure of free pages, and is not affected by the variation of memory utilization.

4.2.2.2.2 Hardware Support for Paging

Hardware support for paging, concentrates on saving the memory necessary for storing of the mapping tables, and on speeding up the mapping of virtual to physical addresses. In principle, each PMT must be large enough to accommodate the maximum size allowed for the address space of a process in a given system. In theory, this may be the entire physical memory. So in a 16 MB system with 256-byte pages, the size of a PMT should be 64k entries. Individual PMT entries are page numbers that are 16 bits long in the sample system, thus requiring 128 KB of physical memory to store a PMT. With one PMT needed for each active process, the total PMT storage can consume a significant portion of physical memory.

Since the actual address space of a process may be well below its allowable maximum, it is reasonable to construct each PMT with only as many entries as its related process has pages. This may be accomplished by means of a

dedicated hardware page-map table limit register (PMTLR). A PMTLR is set to the highest virtual page number defined in the PMT of the running process. Accessing of the PMT of the running process may be facilitated by means of the page-map table base register (PMTBR), which points to the base address of the PMT of the running process. The respective values of these two registers for each process are defined at process-loading time and stored in the related PCB. Upon each process switch, the PCB of the new running process provides the values to be loaded in the PMTBR and PMTLR registers.

Even with the assistance of these registers, address translations in paging systems still require two memory references; one to access the PMT for mapping, and the other to reference the target item in physical memory. To speed it up, a high-speed associative memory for storing a subset of often-used page-map table entries is used. This memory is called the translation look aside buffer (TLB), or mapping cache.

Associative memories can be searched by contents rather than by address. So, the main-memory reference for mapping can be substituted by a TLB reference. Given that the TLB cycle time is very small, the memory-access overhead incurred by mapping can be significantly reduced. The role of the cache in the mapping process is depicted in Figure 6.

As indicated, the TLB entries contain pairs of virtual page numbers and the corresponding page frame numbers where the related pages are stored in physical memory. The page number is necessary to define each particular entry, because a TLB contains only a subset of page-map table entries. Address translation begins by presenting the page-number portion of the virtual address to the TLB. If the desired entry is found in the TLB, the corresponding page frame number is combined with the offset to produce the physical address.

Alternatively, if the target entry is not in TLB, the PMT in memory must be accessed to complete the mapping. This process begins by consulting the PMTLR to verify that the page number provided in the virtual address is within the bounds of the related process's address space. If so, the page number is added to the contents of the PMTBR to obtain the address of the corresponding

PMT entry where the physical page frame number is stored. This value is then concatenated with the offset portion of the virtual address to produce the physical memory address of the desired item.

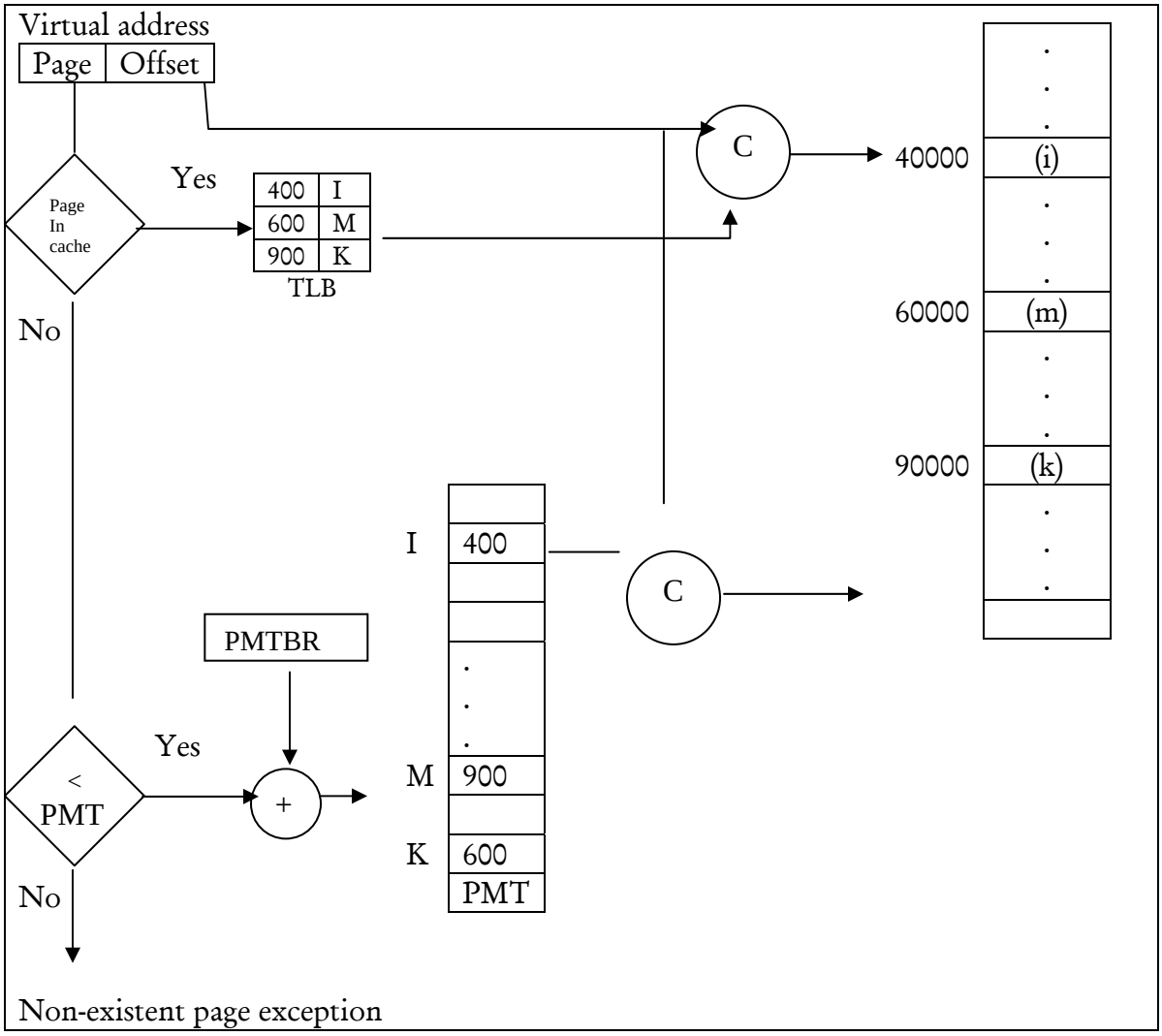


Figure 6 – Translation-lookaside buffer (TLB)

Figure 6 demonstrates that the overhead of TLB search is added to all mappings, regardless of whether they are eventually completed using the TLB or the PMT in main memory. In order for the TLB to be effective, it must satisfy a large portion of all address mappings. Given the generally small size of a TLB because of the high price of associative memories, only the PMT entries most likely to be needed ought to reside in the TLB.

The effective memory-access time, t_{eff} in systems with run-time address translation is the sum of the address translation time, t_{TR} and the subsequent access time needed to fetch the target item from memory, t_M. So $t_{eff} = t_{TR} + t_{M.}$

With TLB used to assist in address translation, t_{TR} becomes

$T_{TR} = h\ t_{TLB} + (1 - h)\ (t_{TLB} + t_M) = t_{TLB} + (1 - h)t_{M;}$ where h is the TLB hit ratio, that is, the ratio of address translations that are contained the TLB over all translations, and thus $0 \leq h \leq 1; t_{TLB}$ is the TLB access time; and t_M is the main-memory access time. Therefore, effective memory-access time in systems with a TLB is $t_{eff} = t_{TLB} + (2 - h)t_{M;}$ It is observed that the hardware used to accelerate address translations in paging systems (TLB) is managed by means of probabilistic algorithms, as opposed to the deterministic mapping-register typing described in relation to segmentation. The reason is that the mechanical splitting of a process's address space into fixed-size chunks produces pages. As a result, a page, unlike a segment, in general does not bear any relationship to the logical entities of the underlying program. For example, a single page may contain a mixture of data, stack, and code. This makes typing and other forms of deterministic loading of TLB entries extremely difficult, in view of the stringent timing restrictions imposed on TLB manipulation.

4.2.2.2.3 Protection and Sharing

Unless specifically declared as shared, distinct address spaces are placed in disjoint areas of physical memory. Memory references of the running process are restricted to its own address space by means of the address translation mechanism, which uses the dedicated PMT. The PMTLR is used to detect and to abort attempts to access any memory beyond the legal boundaries of a process. Modifications of the PMTBR and PMTLR registers are usually possible only by means of privileged instructions, which trap to the OS if attempted in user mode.

By adding the access bits to the PMT entries and appropriate hardware for testing these bits, access to a given page may be allowed only in certain programmer-defined modes such as read-only, execute-only, or other restricted forms of access. This feature is much less flexible in paging systems than segmentation. The primary difference is that paging is supposed to be entirely

transparent to programmers. Mechanical splitting of an address space into pages is performed without any regard for the possible logical relationships between the items under consideration. Since there is no notion of typing, code and data may be mixed within one page. As we shall see, specification of the access rights in paging systems is useful for pages shared by several processes, but it is of much less value inside the boundaries of a given address space.

Protection in paging systems may also be accomplished by means of the protection keys. In principle, the page size should correspond to the size of the memory block protected by the single key. This allows pages belonging to a single process to be scattered throughout memory-a perfect match for paged allocation. By associating access-rights bits with protection keys, access to a given page may be restricted when necessary.

Sharing of pages is quite straightforward with paged memory management. A single physical copy of a shared page can be easily mapped into as many distinct address spaces as desired. Since each such mapping is performed via a dedicated entry in the PMT of the related process, different processes may have different access rights to the shared page. Given that paging is transparent to users, sharing at the page level must be recognized and supported by systems programs. Systems programs must ensure that virtual offsets of each item within a shared page are identical in all participating address spaces.

Like data, shared code must have the same within-page offsets in all address spaces of which it is a part. As usual, shared code that is not executed in mutually exclusive fashion must be reentrant. In addition, unless the shared code is position-independent, it must have the same virtual page numbers in all processes that invoke it. This property must be preserved even in cases when the shared code spans several pages.

4.2.2.3 Virtual Memory

Under Virtual Memory all processes execute code written in terms of virtual addresses that are translated by the memory management hardware into the appropriate physical address. Each process thinks it has access to the whole physical memory of the machine. This solves the relocation problem - no

rewriting of addresses is ever necessary, and the protection problem because a process can no longer express the idea of accessing another process's memory.

Virtual memory allows execution of partially loaded processes. As a consequence, virtual address spaces of active processes in a virtual-memory system can exceed the capacity of the physical memory. This is accomplished by maintaining an image of the entire virtual-address space of a process on secondary storage, and by bringing its sections into main memory when needed. The OS decides which sections to bring in, when to bring them in, and where to place them. Thus, virtual-memory systems provide for automatic migration of portions of address spaces between secondary and primary storage.

Due to the ability to execute a partially loaded process, a process may be loaded into a space of arbitrary size resulting into the reduction of external fragmentation. Moreover, the amount of space in use by a given process may be varied during its memory residence. As a result, the OS may speed up the execution of important processes by allocating them more real memory. Alternatively, by reducing the real-memory holdings of resident processes, the degree of multi-programming can be increased by using the vacated space to activate more processes.

The speed of program execution in virtual-memory systems is bounded from above by the execution speed of the same program run in a non-virtual memory management system. That is due to delays caused by fetching of missing portions of program's address space at run-time.

Virtual memory provides execution of partially loaded programs. But an instruction can be completed only if all code, data, and stack locations that it references reside in physical memory. When there is a reference for an out-of-memory item, the running process must be suspended to fetch the target item from disk. So what is the performance penalty?

An analysis of program behavior provides an answer to the question. Most programs consist of alternate execution paths, some of which do not span the entire address space. On any given run, external and internal program conditions cause only one specific execution path to be followed. Dynamic linking and

loading exploits this aspect of program behavior by loading into memory only those procedures that are actually referenced on a particular run. Moreover, many programs tend to favor specific portions of their address spaces during execution. So it is reasonable to keep in memory only those routines that make up the code of the current pass. When another pass over the source code commences, the memory manager can bring new routines into the main memory and return those of the previous pass back to disk.

4.2.2.3.1 Principles of Operation

Virtual memory can be implemented as an extension of paged or segmented memory management or as a combination of both. Accordingly, address translation is performed by means of PMT, SDT, or both.

The process of address mapping in virtual-memory systems is more formally defined as follows. Let the virtual-address space be V = {0, 1, … , v-1}, and the physical memory space by M = {0, 1, ….m-1}. The OS dynamically allocates real memory to portions of the virtual-address space. The address translation mechanism must be able to associate virtual names with physical locations. At any time the mapping hardware must realize the function f: V $\rightarrow$ M such that

$$f(x) = \begin{cases} r \text{ if item x is in real memory at location r} \\ \text{missing-item exception if item x is not in real memory} \end{cases}$$

Thus, the additional task of address translation hardware in virtual systems is to detect whether the target item is in real memory or not. If the referenced item is in memory, the process of address translation is completed.

We present the operation of virtual memory assuming that paging is the basic underlying memory-management scheme. The detection of missing items is rather straightforward. It is usually handled by adding the presence indicator, a bit, to each entry of PMTs. The presence bit, when set, indicates that the corresponding page is in memory; otherwise the corresponding virtual page is not in real memory. Before loading the process, the OS clears all the presence

bits in the related PMT. As and when specific pages are brought into the main memory, its presence bit is reset.

A possible implementation is illustrated in Figure 7. The presented process's virtual space is assumed to consisting of only six pages. As indicated, the complete process image is present in secondary memory. The PMT contains an entry for each virtual page of the related process. For each page actually present in real memory, the presence bit is set (IN), and the PMT points to the physical frame that contains the corresponding page. Alternatively, the presence bit is cleared (OUT), and the PMT entry is invalid.

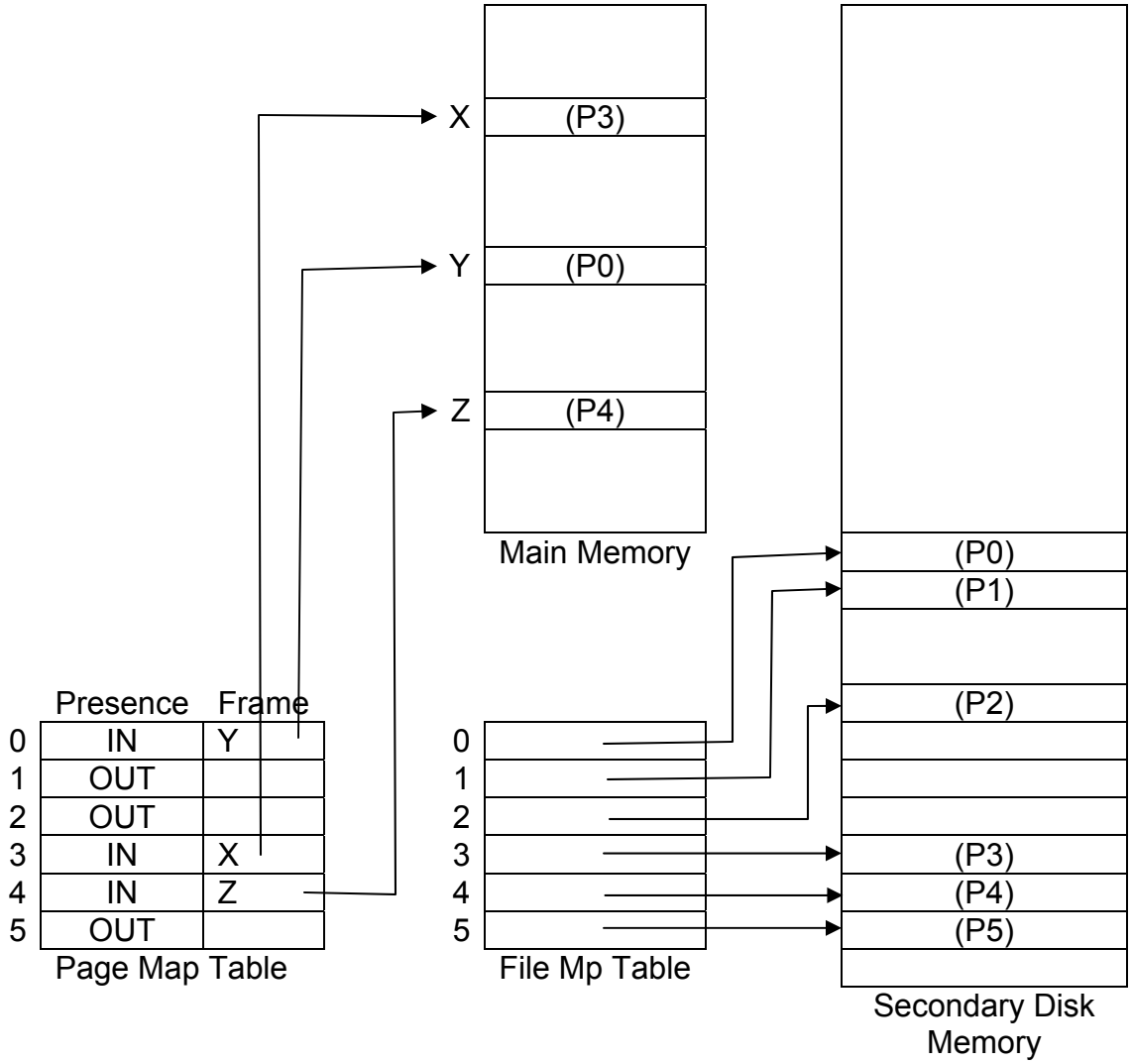


Figure 7: Virtual Memory

The address translation hardware checks the presence bit during the mapping of each memory reference if the bit is set, the mapping is completed as usual.

However, if the corresponding presence bit in the PMT is reset, the hardware generates a missing-item exception known as page fault. When the running process experiences a page fault, it must be suspended until the missing page is brought into main memory.

The disk address of the faulted page is usually provided in the file-map table (FMT). This table is parallel to the PMT. Thus, when processing a page fault, the OS uses the virtual page number provided by the mapping hardware to index the FMT and to obtain the related disk address. A possible format and use of the FMT is depicted in Figure 7.

4.2.2.3.2 Management of Virtual Memory

The implementation of virtual memory requires maintenance of one PMT per active process. Given that the virtual-address space of a process may exceed the capacity of real memory, the size of an individual PMT can be much larger in a virtual than in a real paging system with identical page sizes. The OS maintains one MMT or a free-frame list to keep track of Free/allocated page frames.

A new component of the memory manager's data structures is the FMT. FMT contains secondary-storage addresses of all pages. The memory manager used the FMT to load the missing items into the main memory. One FMT is maintained for each active process. Its base may be kept in the control block of the related process. An FMT has a number of entries identical to that of the related PMT. A pair of page-map table base and page-map length registers may be provided in hardware to expedite the address translation process and to reduce the size of PMT for smaller processes. As with paging, the existence of a TLB is highly desirable to reduce the negative effects of mapping on the effective memory bandwidth.

The allocation of only a subset of real page frames to the virtual-address space of a process requires the incorporation of certain policies into the virtual-memory manager. We may classify these policies as follows:

1. Allocation policy: How much real memory to allocate to each active process

2. Fetch policy: Which items to bring and when to bring them from secondary storage into the main memory
3. Replacement policy: When a new item is to be brought in and there is no free real memory, which item to evict in order to make room.
4. Placement policy: Where to place an incoming item

4.2.2.3.3 Program Behavior

By minimizing the number of page faults, the effective processor utilization, effective disk I/O bandwidth, and program turnaround times may be improved. It is observed that there is a strong tendency of programs to favor subsets of their address spaces during execution. This phenomenon is known as locality of reference. Both temporal and spatial locality of reference has been observed.

(a) Spatial locality suggests that once an item is referenced, there is a high probability that it or its neighboring items are going to be referenced in the near future.
(b) Temporal locality is the tendency for a program to reference the same location or a cluster several times during brief intervals of time. Temporal locality of reference is exhibited by program loops.

Both temporal and spatial locality of reference is dynamic properties in the sense that the identity of the particular pages that compose the actively used set varies with time. As observed, the executing program moves from one locality to another in the course of its execution. Statistically speaking, the probability that a particular memory reference is going to be made to a specific page is a time-varying function. It increases when pages in its current locality are being referenced, and it decreases otherwise. The evidence also suggests that the executing program moves slowly from one locality to another. Locality of reference basically suggests that a significant portion of memory references of the running process may be made to a subset of its pages. These findings may be utilized for implementation of replacement and allocation policies.

4.2.2.3.4 Replacement Policies

If a page fault is there, then it is to be brought into the main memory necessitating creation of a room for it. There are two options for this situation:

- The faulted process may be suspended until availability of memory.
- A page may be removed to make room for the incoming one.

Suspending a process is not an acceptable solution. Thus, removal is commonly used to free the memory needed to load the missing items. A replacement policy decides the victim page for eviction. In virtual memory systems all pages are kept on the secondary storage. As and when needed, some of those pages are copied into the main memory. While executing, the running process may modify its data or stack areas, thus making some resident pages different from their disk images (dirty page). So it must be written back to disk in place of its obsolete copy. When a page that has not been modified (clean page) during its residence in memory is to be evicted, if can simply be discarded. Tracking of page modifications is usually performed in hardware by adding a written-into bit called as dirty bit, to each entry of the PMT. It indicates whether the page is dirty or clean.

4.2.2.3.5 Replacement Algorithms

First-In-First-Out (FIFO):

The FIFO algorithm replaces oldest pages i.e. the resident page that has spent the longest time in memory. To implement the FIFO page-replacement algorithm, the memory manager must keep track of the relative order of the loading of pages into the main memory. One way to accomplish this is to maintain a FIFO queue of pages.

FIFO fails to take into account the pattern of usage of a given page; FIFO tends to throw away frequently used pages because they naturally tend to stay longer in memory. Another problem with FIFO is that it may defy intuition by increasing the number of page faults when more real pages are allocated to the program. This behavior is known as Belady's anomaly.

Least Recently Used (LRU):

The LRU algorithm replaces the least recently used resident page. LRU algorithm performs better than FIFO because it takes into account the patterns of

program behavior by assuming that the page used in the most distant past is least likely to be referenced in the near future. The LRU algorithm belongs to a larger class of stack replacement algorithms. A stack algorithm is distinguished by the property of performing better, or at least not worse, when more real memory is made available to the executing program. Stack algorithms therefore do not suffer from Belady's anomaly.

The implementation of the LRU algorithm imposes too much overhead to be handled by software alone. One possible implementation is to record the usage of pages by means of a structure similar to the stack. Whenever a resident page is referenced, it is removed from its current stack position and placed at the top of the stack. When a page eviction is in order, the page at the bottom of the stack is removed from memory.

Maintenance of the page-referencing stack requires it’s updating for each page reference, regardless of whether it results in a page fault or not. So the overhead of searching the stack, moving the reference page to the top, and updating the rest of the stack accordingly must be added to all memory references. But the FIFO queue needs to be updated only when page faults occur-overhead almost negligible in comparison to the time required for processing of a page fault.

Optimal (OPT):

The algorithm by Belady, removes the page to be reference in the most distant future i.e. page out the page that will be needed the furthest in the future. This is impossible (halting problem), but provides an interesting benchmark. Since it requires future knowledge, the OPT algorithm is not realizable. Its significance is theoretical, as it can serve as a yardstick for comparison with other algorithms.

Approximations-Clock:

One popular algorithm combines the relatively low overhead of FIFO with tracking of the resident-page usage, which accounts for the better performance of LRU. This algorithm is sometimes referred to as Clock, and it is also known as not recently used (NRU). The algorithm makes use of the referenced bit, which is associated with each resident page. The referenced bit is set whenever the related page is reference and cleared occasionally by software. Its setting

indicates whether a given page has been referenced in the recent past. How recent this past is depends on the frequency of the referenced-bit resetting. The page-replacement routine makes use of this information when selecting a victim for removal.

The algorithm is usually implemented by maintaining a circular list of the resident pages and a pointer to the page where it left off. The algorithm works by sweeping the page list and resetting the presence bit of the pages that it encounters. This sweeping motion of the circular list resembles the movement of the clock hand, hence the name clock. The clock algorithm seeks and evicts pages not recently used in order to free page frames for allocation to demanding processes. When it encounters a page whose reference bit is cleared, which means that the related page has not been referenced since the last sweep, the algorithm acts as follows:

(1) If the page is modified, it is marked for clearing & scheduled for writing to disk.

(2) If the page is not modified, it is declared non-resident, and the page frames that it occupies are feed.

The algorithm continues its operation until the required numbers of page frames are freed. The algorithm may be invoked at regular time intervals or when the number of free page frames drops below a specified threshold.

Other approximations and variations on this theme are possible. Some of them track page usage more accurately by means of a reference counter that counts the number of sweeps during which a given page is found to be un-referenced. Another possibility is to record the states of referenced bits by shifting them occasionally into related bit arrays. When a page is to be evicted, the victim is chosen by comparing counters or bit arrays in order to find the least frequently reference page. The general idea is to devise an implementable algorithm that bases its decisions on measured page usage and thus takes into account the program behavior patterns.

4.2.2.3.6 Allocation Policies

The allocation policy must compromise among conflicting requirements such as:

(a) Reduced page-fault frequency,

(b) Improved turn-around time,

(c) Improved processor utilization, etc.

Giving more real pages to a process will result in reduced page-fault frequency and improved turnaround time. But it reduces the number of active processes that may coexist in memory at a time resulting into the lower processor utilization factor. On the other hand, if too few pages are allocated to a process, its page-fault frequency and turnaround times may deteriorate.

Another problem caused by under-allocation of real pages may be encountered in systems that opt for restarting of faulted instructions. If fewer pages are allocated to a process than are necessary for execution of the restartable instruction that causes the largest number of page faults in a given architecture, the system might fault continuously on a single instruction and fail to make any real progress.

Consider a two-address instruction, such as Add @X, @Y, where X and Y are virtual addresses and @ denotes indirect addressing. Assuming that the operation code and operand addresses are encoded in one word each, this instruction need three words for storage. With the use of indirect addressing, eight memory references are needed to complete execution of this instruction: three to fetch the instruction words, two to fetch operand addresses, two to access the operands themselves (indirect addressing), and one to store the result. In the worst case, six different pages may have to reside in memory concurrently in order to complete execution of this instruction: two if the instruction crosses a page boundary, two holding indirect addresses, and two holding the target operands. A likely implementation of this instruction calls for the instruction to be restarted after a page fault. If so, with fewer than six pages allocated to the process that executes it, the instruction may keep faulting forever. In general, the lower limit on the number of pages imposed by the described problem is architecture-dependent. In any particular implementation,

the appropriate bound must be evaluated and built into the logic of the allocation routine.

While we seem to have some guidance as to the minimal number of pages, the reasonable maximum remains elusive. It is also unclear whether a page maximum should be fixed for a given system or determined on an individual basis according to some specific process attributes. Should the maximum be defined statically or dynamically, in response to system resource utilization and availability, and perhaps in accordance with the observable behavior of the specific process?

From the allocation module's point of view, the important conclusion is that each program has a certain threshold regarding the proportion of real to virtual pages, below which the number of page faults increases very quickly. At the high end, there seems to be a certain limit on the number of real pages, above which an allocation of additional real memory results in little or in moderate performance improvement. Thus, we want to allocate memory in such a way that each active program is between these two extremes.

Being program-specific, the upper and lower limits should probably not be fixed but derived dynamically on the basis of the program faulting behavior measured during its execution. When resource utilization is low, activating more processes may increase the degree of multiprogramming. However, the memory manager must keep track of the program behavior when doing so. A process that experiences a large number of page faults should be either allocated more memory or suspended otherwise. Likewise, a few pages may be taken away from a process with a low page-fault rate without great concern. In addition, the number of pages allocated to a process may be influenced by its priority (higher priority may indicate that shorter turnaround time is desirable), the amount of free memory, fairness, and the like.

Thrashing: Although the complexity and overhead of memory allocation should be within a reasonable bound, the use of oversimplified allocation algorithms has the potential of crippling the system throughput. If real memory is over-allocated to the extent that most of the active programs are above their upper page-fault-

rate thresholds, the system may exhibit a behavior known as thrashing. With very frequent page faults, the system spends most of its time shuttling pages between main memory and secondary memory. Although the disk I/O channel may be overloaded by this activity, but processor utilization is reduced.

One way of introducing thrashing behavior is dangerously logical and simple. After observing a low processor utilization factor, the OS may attempt to improve it by activating more processes. if no free pages are available, the holdings of the already-active processes may be reduced. This may drive some of the processes into the high page-fault zone. As a result, the processor utilization may drop while the processes are awaiting their pages to be brought in. In order to improve the still-decreasing processor utilization, the OS may decide to increase the degree of multi-programming even further. Still more pages will be taken away from the already-depleted holdings of the active processes, and the system is hopelessly on its way to thrashing. It is obvious that global replacement strategies are susceptible to thrashing.

Thus a good design must make sure that the allocation algorithm is not unstable and inclined toward thrashing. Knowing the typical patterns of program behavior, we want to ensure that no process is allocated too few pages for its current needs. Too few pages may lead to thrashing, and too many pages may unduly restrict the degree of multi-programming and processor utilization.

Page-Fault Frequency (PFF)

This policy uses an upper and lower page-fault frequency threshold to decide for allocation of new page frames. The PFF parameter P may be defined as: P = 1/T

Where T is the critical inter-page fault time. P is usually measured in number of page faults per millisecond. The PFF algorithm may be implemented as follows:

1. The OS defines a system-wide (or per-process) critical page-fault frequency, P.
2. The OS measures the virtual (process) time and stores the time of the most recent page fault in the related process control block.

When a page fault occurs, the OS acts as follows:

- If the last page fault occurred less than T = 1/P ms ago, the process is operating above the PFF threshold, and a new page frame is added from the pool to house the needed page.
- Otherwise, the process is operating below the PFF threshold P, and a page frame occupied by a page whose reference bit and written-into bit are not set is freed to accommodate the new page.
- The OS sweeps and resets referenced bits of all resident pages. Pages that are found to be unused, unmodified, and not shared since the last sweep are released, and the freed page frames are returned to the pool for future allocations.

For completeness, some policies need to be employed for process activation and deactivation to maintain the size of the pool of free page frames within desired limits.

4.2.2.3.7 Hardware Support and Considerations

Virtual memory requires:

(1) instruction interruptibility and restartability,
(2) a collection of page status bits associated with each page descriptor,
(3) And if based on paging - a TLB to accelerate address translations.

Choice of the page size can have a significant impact on performance of a virtual-memory system. In most implementations, one each of the following bits is provided in every page descriptor:

- Presence bit, used to aid detection of missing items by the mapping hardware
- Written-into (modified) bit, used to reduce the overhead incurred by the writing of unmodified replaced pages to disk
- Referenced bit, used to aid implementation of the replacement policy

An important hardware accelerator in virtual-memory systems is the TLB. Although system architects and hardware designers primarily determine the details of the TLB operation, the management of TLB is of interest because it deals with problems quite similar to those discussed in the more general

framework of virtual memory. TLB hardware must incorporate allocation and replacement policies so as to make the best use of the limited number of mapping entries that the TLB can hold. An issue in TLB allocation is whether to devote all TLB entries to the running process or to distribute them somehow among the set of active processes. The TLB replacement policy governs the choice of the entry to be evicted when a miss occurs and another entry needs to be brought in.

Allocation of all TLB entries to the running process can lead to relatively lengthy initial periods of "loading" the TLB whenever a process is scheduled. This can lead to the undesirable behavior observed in some systems when an interrupt service routine (ISR) preempts the running process. Since a typical ISR is only a few hundred instructions long, it may not have enough time to load the TLB. This can result in slower execution of the interrupt service routine due to the need to reference PMT in memory while performing address translations. Moreover, when the interrupted process is resumed, its performance also suffers from having to load the TLB all over again. One way to combat this problem is to use multi-context TLBs that can contain and independently manage the PMT entries of several processes. With a multi-context TLB, when a process is scheduled for execution, it may find some of its PMT entries left over in the TLB from the preceding period of activity. Management of such TLBs requires the identity of the corresponding process to be associated with each entry, in order to make sure that matches are made only with the TLB entries belonging to the process that produced the addresses to be mapped.

Removal of TLB entries is usually done after each miss. If PMT entries of several processes are in the buffer, the victim may be chosen either locally or globally. Understandably, some preferential treatment is usually given to holdings of the running process. In either case, least recently used is a popular strategy for replacement of entries.

The problem of maintaining consistency between the PMT entries and their TLB copies in the presence of frequent page moves must also be tackled by hardware designers. Its solution usually relies on some specialized control instructions for TLB flushing or for it selective invalidation.

Another hardware-related design consideration in virtual-memory systems is whether I/O devices should operate with real or virtual addresses.

A hardware/software consideration involved in the design of paged systems Is the choice of the page size. Primary factors that influence this decision are

(1) Memory utilization and cost.

(2) Page-transport efficiency.

Page-transport efficiency refers to the performance cost and overhead of fetching page from the disk or, in a diskless workstation environment, across the network. Loading of a page from disk consists of two basic components: the disk-access time, and the page-transfer time. Head positioning delays generally exceed disk-memory transfer times by order of magnitude. Thus, total page-transfer time tends to be dominated by the disk positioning delay, which is independent of the page size.

Small page size reduces page breakage, and it may make better use of memory by containing only a specific locality of reference. On the other hand, small pages may result in excessive size of mapping tables in virtual systems with large virtual-address spaces. Page-transport efficiency is also adversely affected by small page sizes, since the disk-accessing overhead is imposed for transferring a relatively small group of bytes.

Large pages tend to reduce table fragmentation and to increase page-transport efficiency. This is because the overhead of disk accessing is amortized over a larger number of bytes whenever a page is transferred between disk and memory. On the negative side, it may impact memory utilization by increasing page breakage and by spanning more than one locality of reference. If multiple localities contained in a single page have largely dissimilar patterns of reference, the system may experience reduced effective memory utilization and wasted I/O bandwidth.

4.2.2.3.8 Protection and Sharing

The frequent moves of items between main and secondary memory may complicate the management of mapping tables in virtual systems. When several

parties share an item in real memory, the mapping tables of all involved processes must point to it. If the shared item is selected for removal, all concerned mapping tables must be updated accordingly. The overhead involved tends to outweigh the potential benefit of removing shared items. Many systems simplify the management of mapping tables by fixing the shared objects in memory. An interesting possibility provided by large virtual-address spaces is to treat the OS itself as a shared object. As such, the OS is mapped as a part of each user's virtual space. To reduce table fragmentation, dedicated mapping registers are often provided to access a single physical copy of the page-map table reserved for mapping references to the OS. One or more status bits direct the mapping hardware to use the public or private mapping table, as appropriate for each particular memory reference. In this scheme, different users have different access rights to portions of the OS. Moreover, the OS-calling mechanism may be simplified by avoiding expensive mode switches between users and the OS code. With the protection mechanism provided by mapping, a much faster CALL instruction, or its variant, may be used to invoke the OS.

4.2.2.4 Segmentation and Paging

It is also possible to implement virtual memory in the form of demand segmentation inheriting the benefits of sharing and protection provided by segmentation. Moreover, their placement policies are aided by explicit awareness of the types of information contained in particular segments. For example, a "working set" of segments should include at least one each of code, data, and stack segments. As with segmentation, inter-segment references alert the OS to changes of locality. However, the variability of segment sizes and the within-segment memory contiguity requirement complicate the management of both main and secondary memories. Placement strategies are quite complex in

segmented systems. Moreover, allocation and deallocation of variable-size storage areas to hold individual segments on disk imposes considerably more overhead than handling of pages that are usually designed to fit in a single disk block.

On the other hand, paging is very easy for the management of main and secondary memories, but it is inferior with regard to protection and sharing. The transparency of paging necessitates the use of probabilistic replacement algorithms which virtually no guidance from users, they are forced to operate mainly on the basis of their observations of program behavior.

Both segmented and paged implementations of virtual memory have their advantages/disadvantages. Some systems combine the two approaches in order to enjoy the benefits of both. One approach is to use segmentation from the user's point of view but to divide each segment into pages of fixed size for purposes of allocation. In this way, the combined system retains most of the advantages of segmentation. At the same time, the problems of complex segment placement and management of secondary memory are eliminated by using paging. The principle of address translation in combined segmentation and paging systems is shown in Figure 8. Both segment descriptor tables and PMT are required for mapping. Instead of containing the base and limit of the corresponding segment, each entry of the SDT contains the base address and size of the PMT to be used for mapping of the related segment's pages. The presence bit in each PMT entry indicates availability of the corresponding page in the real memory. Access rights are recorded as a part of segment descriptors, although they may be placed or refined in the entries of the PMT. Each virtual address consists of three fields: segment number, page number, and offset within the page. When a virtual address is presented to the mapping hardware, the segment number is used to locate the corresponding PMT. Provided that the issuing process is authorized to make the intended type of reference to the target segment, the page number is used to index the PMT. If the presence bit is set, obtaining the page-frame address from the PMT and combining this with the offset part of the virtual address complete the mapping. If the target page is

absent from real memory, the mapping hardware generates a page-fault exception, which is processed. At both mapping stages, the length fields are used to verify that the memory references of the running process lie within the confines of it address space.

Many variations of this powerful scheme are possible. For example, the presence bit may be included with entries of the SDT. It may be cleared when no pages of the related segment are in real memory. When such a segment is referenced, bringing several of lits pages into main memory may process the segment fault. In general, page re-fetching has been more difficult to implement in a way that performs better than demand paging. One of the main reasons for this is the inability to predict the use of previously un-referenced pages. However, referencing of a particular segment increases the probability of its constituent pages being referenced.

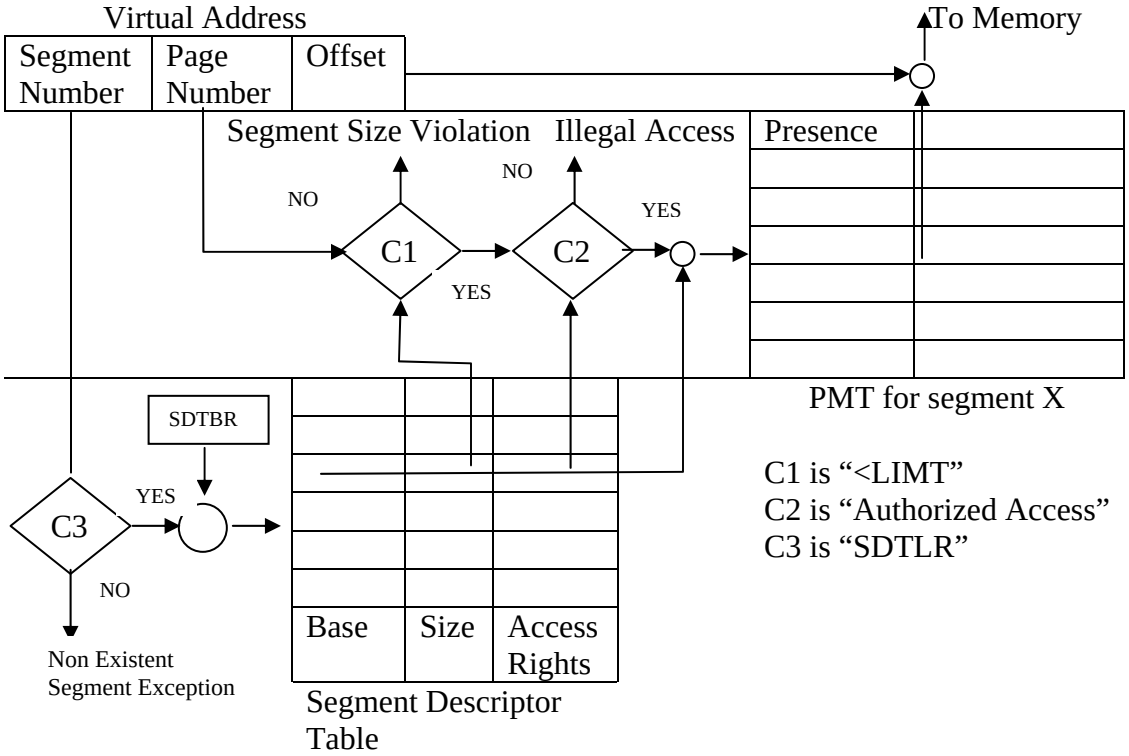


Figure 8 – Segmentation and paging

While the combination of segmentation and paging is certainly appealing, it requires two memory accesses to complete the mapping of each virtual address resulting into the reduction of the effective memory bandwidth by two-thirds.

4.3 Summary

Segmentation allows breaking of the virtual address space of a single process into separate entities that may be placed in noncontiguous areas of physical memory. As a result, the virtual-to-physical address translation at instruction execution time in such systems is more complex, and some dedicated hardware support is necessary to avoid a drastic reduction in effective memory bandwidth. Since average segment sizes are usually smaller then average process sizes, segmentation can reduce the impact of external fragmentation on the performance of systems with dynamically partitioned memory. Other advantages of segmentation include dynamic relocation, finely grained protection both within and between address spaces, ease of sharing, and facilitation of dynamic linking and loading.

No doubt segmentation reduces the impact of fragmentation and offers superior protection and sharing by dividing each process's address space into logically related entities that may be placed into non-contiguous areas of physical memory. But paging simplifies allocation and de-allocation of memory by dividing address spaces into fixed-sized chunks. Execution-time translation of virtual to physical addresses, usually assisted by hardware, is used to bridge the gap between contiguous virtual addresses and non-contiguous physical addresses where different pages may reside.

It is very desirable to execute a process whose logical address space is larger than the available physical address space and the option is virtual memory. Virtual memory removes the restriction on the size of address spaces of individual processes that is imposed by the capacity of the physical memory installed in a given system. In addition, virtual memory provides for dynamic migration of portions of address spaces between primary and secondary memory in accordance with the relative frequency of usage.

If the total memory requirement is larger than the available physical memory, then memory management system has to create the house for new pages by replacing some pages from the memory. A number of page replacement policies have been proposed such as FIFO, LRU, NRU, etc with their merits and

demerits. FIFO implementation is easy but suffers from Belady anomaly. Optimal replacement requires future knowledge. LRU is n approximation of optimal but difficult to implement. After page replacement, there is the need for frame allocation policy. An improper allocation policy may result into thrashing.

It is also possible to implement virtual memory in the form of demand segmentation inheriting the benefits of sharing and protection provided by segmentation but placement strategies are complex, allocation and deallocation of variable-size storage areas to hold individual segments on disk imposes more overhead. On the other hand, paging is very easy for the management of main and secondary memories, but it is inferior with regard to protection and sharing. Some systems combine the two approaches in order to enjoy the benefits of both.

4.4 Keywords

External Fragmentation is waste of memory between partitions caused by scattered non-contiguous free space.

Internal fragmentation is waste of memory within a partition caused by difference between size of partition and the process allocated.

Page: The virtual-address space of a process is divided into fixed-size blocks of the same size, called pages.

Page fault: The phenomenon of not finding a referenced page in the memory is known a page fault.

TLB (Translation Look aside Buffer): It is a high-speed associative memory, used to speed up memory access, by for storing a subset of often-used page-map table entries.

PMT (Page Map Table): It is a table used to translate a virtual address into actual physical address in paging system.

Locality of Reference: There is a strong tendency of programs to favor subsets of their address spaces during execution. This phenomenon is known as locality of reference.

4.5 Self Assessment Questions (SAQ)

1. Define segmentation and write a detailed note on the address translation mechanism in segmentation.
2. Write a detailed mote on sharing in segmentation. How the access rights are implementation in sharing in segmentation.
3. What is the basic difference between paging and segmentation? Which one is better and why?
4. What is Table Look aside Buffer (TLB)? How is it used to speed up the memory access? Explain.
5. Write short notes on following:
   (a) Thrashing
   (b) Page fault frequency
   (c) Sharing in virtual memory
6. Differentiate between following:
   a. Dirty page and clean page
   b. Logical Address and Physical Address
   c. Spatial and temporal locality of reference
   d. Segmentation and paging
7. What is the common drawback of all the real memory management techniques? How is it overcome in virtual memory management schemes?
8. What extra hardware do we require for implementing demand paging and demand segmentation?
9. Show that LRU page replacement policy possesses the stack property.
10. Differentiate between internal and external fragmentation.
11. What do you understand by thrashing? What are the factors causing it?
12. Compare FIFO page replacement policy with LRU page replacement on the basis of overhead.

4.6 Suggested Readings / Reference Material

11. Operating Systems Concepts, 5th Edition, Silberschatz A., Galvin P.B., John Wiley and Sons.

12. Systems Programming and Operating Systems, 2nd Revised Edition, Dhamdhere D.M., Tata McGraw Hill Publishing Company Ltd., New Delhi.
13. Operating Systems, Madnick S.E., Donovan J.T., Tata McGraw Hill Publishing Company Ltd., New Delhi.
14. Operating Systems-A Modern Perspective, Gary Nutt, Pearson Education Asia, 2000.
15. Operating Systems, Harris J.A., Tata McGraw Hill Publishing Company Ltd., New Delhi, 2002.

Lesson number: 5	***Writer: Dr. Rakesh Kumar***
File System - I	***Vetter: Dr. Pradeep Bhatia***

5.0 Objectives

A file is a logical collection of information and file system is a collection of files. The objective of this lesson is to discuss the various concepts of file system and make the students familiar with the different techniques of file allocation and access methods. We also discuss the ways to handle file protection, which is necessary in an environment where multiple users have access to files and where it is usually desirable to control by whom and in what ways files may be accessed.

5.1 Introduction

The file system is the most visible aspect of an operating system. While the memory manager is responsible for the maintenance of primary memory, the file manager is responsible for the maintenance of secondary storage (e.g., hard disks). It provides the mechanism for on-line storage of and access to both data and programs of the operating system and all the users of the computer system. The file system consists of two distinct parts: a collection of files, each storing related data and a directory structure, which organizes and provides information about all the files in the system. Some file systems have a third part, partitions, which are used to separate physically or logically large collections of directories.

Nutt describes the responsibility of the file manager and defines the file, the fundamental abstraction of secondary storage:

> *"Each file is a named collection of data stored in a device. The file manager implements this abstraction and provides directories for organizing files. It also provides a spectrum of commands to read and write the contents of a file, to set the file read/write position, to set and use the protection mechanism, to change the ownership, to list files in a directory, and to remove a file...The file manager provides a protection mechanism to allow machine users to administer how processes executing on behalf of different users can access the*

information in files. File protection is a fundamental property of files because it allows different people to store their information on a shared computer, with the confidence that the information can be kept confidential."

5.2 Presentation of Contents

5.2 PRESENTATION OF CONTENTS

5.2.1 FILE CONCEPTS

The most important function of an operating system is the effective management of information. The modules of the operating system dealing with the management of information are known as file system. The file system provides the mechanism for online storage and access to both data and programs. The file system resides permanently on secondary storage, which has the main requirement that it must be able to hold a large amount of data, permanently. The desirable features of a file system are:

4. Minimal I/O operations.
5. Flexible file naming facilities.
6. Automatic allocation of file space.
7. Dynamic allocation of file space.
8. Unrestricted flexibility between logical record size and physical block size.
9. Protection of files against illegal forms of access.
10. Static and dynamic sharing of files.
11. Reliable storage of files.

This lesson is primarily concerned with issues concerning file storage and access on the most common secondary storage medium, the disk.

A file is a collection of related information units (records) treated as a unit. A record is itself a collection of related data elements (fields) treated as a unit. A field contains a single data item.

So file processing refers to reading/writing of records in a file and processing of the information in the fields of a record.

5.2.1.1 File operations

File operations are generally simple and intuitive set of operations on files. Not all of these are supported by all file systems. In some file systems, operations are implemented in terms of other file operations. Major file operations performed are as follows:

- Open: This lets the Operating Systems know that the current process will be interested in a file soon. In some sense, it's extraneous, but in cases where the file resides on a medium that has significant startup cost not returning the open call until the file is ready for access is a good idea.
- Close: Let the Operating Systems know that the process is done with this file, and that the Operating Systems can reclaim the resources allocated to manipulating the file. (The data and meta-data are updated, but remain in the file system, of course.) Some systems delay writes or cache data for future reads. Close is an indication to them that pending writes must be flushed and that cached reads can be discarded.
- Read operation: This operation read information contained in the file.
- Write operation: This operation write new information into a file at any point or

overwriting existing information in a file. Strictly this means to change existing data in the file to a new value, but many systems also use the write system call to append data.

- Deleting file: Delete a file and release its storage space for use in other files.
- Appending file: Write new information at the end of a file. This shares aspects with write, but includes the idea that the underlying file is changing size. Append means that the Operating Systems must increase the allocation of storage to the file. Some Operating Systems determine whether a write system call causes an append or a write based on the current byte and the length of the buffer written.
- Seek: Files have a notion of the current byte (or record) of the file that will next be accessed. On files that can be randomly accessed, seek allows the calling process to set the current byte (or record).
- Renaming file: Change a name of the file in the file system. This may be an operation on the file or a directory depending on the file system.
- Creating file: Create a new file. This may allocate space for the file, or just reserve the name for future action.
- Moving file: To move the file from one place to another.
- Sorting file: To arrange the data in file in some order.
- Execute a file
- Coping file
- Merging files
- Comparing file

5.2.1.2 File Naming

Each file is a distinct entity and therefore a naming convention is required to distinguish one from another. The operating systems generally employ a naming system for this purpose. In fact, there is a naming convention to identify each resource in the computer system and not files alone.

5.2.1.3 File Types

The files under UNIX can be categorized as follows:

- Ordinary files.

- Directory files.
- Special files.
- FIFO files.

Ordinary Files

Ordinary files are the one, with which we all are familiar. They may contain executable programs, text or databases. You can add, modify or delete them or remove the file entirely.

Directory Files

Directory files, as discussed earlier also represent a group of files. They contain list of file names and other information related to these files. Some of the commands, which manipulate these directory files, differ from those for ordinary files.

Special Files

Special files are also referred to as device files. These files represent physical devices such as terminals, disks, printers and tape-drives etc. These files are read from or written into just like ordinary files, except that operation on these files activates some physical devices. These files can be of two types (i) character device files and (ii) block device file. In character device files data are handled character by character, as in case of terminals and printers. In block device files, data are handled in large chunks of blocks, as in the case of disks and tapes.

FIFO Files

FIFO (first-in-first-out) are files that allow unrelated processes to communicate with each other. They are generally used in applications where the communication path is in only one direction, and several processes need to communicate with a single process. For an example of FIFO file, take the pipe in UNIX. This allows transfer of data between processes in a first-in-first-out manner. A pipe takes the output of the first process as the input to the next process, and so on.

5.2.1.4 Symbolic Link

A link is effectively a pointer or an alias to another file or subdirectory. For example, a link may be implemented as an absolute or relative path name (a symbolic link). When a reference to a file is made, we search the directory. The directory entry is marked as a link and the name of the real file (or directory) is given. We resolve the link by using the path name to locate the real file. Links are easily identified by their format in the directory entry (or by their having a special type on systems that support types), and are effectively named indirect pointers.

A symbolic link can be deleted without deleting the actual file it links. There can be any number of symbolic links attached to a single file.

Symbolic links are helpful in sharing a single file called by different names. Each time a link is created, the reference count in its inode is incremented by one. Whereas deletion of link decreases the reference count by one. The operating system denies deletion of such files whose reference count is not 0, thereby meaning that the file is in use.

In a system where sharing is implemented by symbolic links, this situation is somewhat easier to handle. The deletion of a link does not need to affect the original file; only the link is removed. If the file entry itself is deleted, the space for the file is deallocated, leaving the links dangling. We can search for these links and remove them also, but unless a list of the associated link is kept with each file, this search can be expensive. Alternatively, we can leave the links until an attempt is made to use them. At that time, we can determine that the file of the name given by the link does not exist, and can fail to resolve the link name; the access is treated just like any other illegal file name. (In this case, the system designer should consider carefully what to do when a file is deleted and another file of the same name is created, before a symbolic link to the original file is used.) In the case of UNIX, symbolic links are left when a file is deleted, and it is up to the user to realize that the original file is gone or has been replaced.

Another approach to deletion is to preserve the file until all references to it are deleted. To implement this approach, we must have some mechanism for determining that the last reference to the file has been deleted. We could keep a list of all references to a file (directory entries or symbolic links). When a link or a copy of the directory entry is established, a new entry is added to the file-reference list. When a link or directory entry is deleted, we remove its entry on the list. The file is deleted when its file-reference list is empty.

The trouble with this approach is the variable and potentially large size of the file-reference list. However, we really do not need to keep the entire list - we need to keep only a count of the number of references. A new link or directory entry increments the reference counts; deleting a link or entry decrements the count. When the count is 0, the file can be deleted; there are no remaining references to it. The UNIX operating system uses this approach for non-symbolic links, or hard links, keeping a reference count in the file information block or

inode). By effectively prohibiting multiple references to directories, we maintain an acyclic-graph structure.

To avoid these problems, some systems do not allow shared directories links. For example, in MS-DOS, the directory structure is a tree structure.

5.2.1.5 File Sharing and Locking

The owner of a file uses the access control list of the file to authorize some other users to access the file. In a multi-user environment a file is required to be shared among more than one user. There are several techniques and approaches to affect this operation. File sharing can occur in two modes (i) sequential sharing and (ii) concurrent sharing. Sequential sharing occurs when authorized users access a shared file one after another. So any change made by a user is reflected to other users also. Concurrent sharing occurs when two or more users access a file over the same period of time. Concurrent sharing may be implemented in one of the following three forms:

(a) Concurrent sharing using immutable files: In it any program cannot modify the file being shared.

(b) Concurrent sharing using single image mutable files: An image is a view of a file. All programs concurrently sharing the file see the same image of the file. So changes made by one program are also visible to other programs sharing the file.

(c) Concurrent sharing using multiple image mutable files: Each program accessing the file has its own image of the file. So many versions of the file at a time may exist and updates made by a user may not be visible to some concurrent user.

There are three different modes to share a file:

- Read only: In this mode the user can only read or copy the file.
- Linked shared: In this mode all the users sharing the file can make changes in this file but the changes are reflected in the order determined by the operating systems.
- Exclusive mode: In this mode a single user who can make the changes (while others can only read or copy it) acquires the file.

Another approach is to share a file through symbolic links. This approach poses a couple of problems - concurrent updation problem, deletion problem. If two users try to update the same file, the updating of one of them will be reflected at a time. Besides, another user must not delete a file while it is in use.

File locking gives processes the ability to implement mutually exclusive access to a file. Locking is mechanism through which operating systems ensure that the user making changes to the file is the one who has the lock on the file. As long as the lock remains with this user, no other user can alter the file. Locking can be limited to files as a whole or parts of a file. Locking may apply to any access or different levels of locks may exist such as read/write locks etc.

5.2.1.6 File-System Structure

Disks provide the bulk of secondary storage on which a file system is maintained. To improve I/O efficiency, I/O transfers between memory and disks are performed in units of blocks. Each block is one or more sectors. Depending on the disk drive, sectors vary from 32 bytes to 4096 bytes; usually, they are 512 bytes. The blocking method determines how a file's records are allocated into blocks:

Fixed blocking: An integral number of fixed-size records are stored in each block. No record may be larger than a block.

Unspanned blocking: Multiple variable size records can be stored in each block but no record may span multiple blocks.

Spanned blocking: Records may be stored in multiple blocks. There is no limit on the size of a record.

Disks have two important characteristics that make them a convenient medium for storing multiple files:

(a) They can be rewritten in place; it is possible to read a block from the disk, to modify the block, and to write it back into the same place.

(b) One can access directly any given block of information on the disk. Thus, it is simple to access any file either sequentially or randomly, and switching from one file to another added requires only moving the read-write heads and waiting for the disk to rotate.

To provide an efficient and convenient access to the disk, the operating system imposes a file system to allow the data to be stored, located, and retrieved easily. A file system poses two quite different design problems.

(a) How the file system should look to the user? This task involves the definition of a file and its attributes, operations allowed on a file and the directory structure for organizing the files.

(b) Algorithms and data structure must be created to map the logical file system onto the physical secondary storage devices.

5.2.1.7 File-System Mounting

Just as a file must be opened before it is used, a file system must be mounted before it can be available to processes on the system. The mount procedure is straightforward. The operating system is given the name of the device and the location within the file structure at which to attach the file system (called the mount point). For instance, on the UNIX system, a file system containing user's home directory might be mounted as /home; then, to access the directory structure within that file system, one could precede the directory names with /home, as in /home/jane. Mounting that file system under /users would result in the path name /users/jane to reach the same directory.

Next, the operating system verifies that the device contains a valid file system. It does so by asking the device driver to read the device directory and verifying that the directory has the expected format. Finally, the operating system notes its directory structure that a file system is mounted at the specified mount point. This scheme enables the operating system to traverse its directory structure, switching among file systems as appropriate. Consider the actions of the Macintosh Operating System.

Whenever the system encounters a disk for the first time (hard disks are found at boot time, floppy disks ate seen when they are inserted into the drive), the Macintosh Operating System searches for a file system on the device. If it finds one, it automatically mounts the file system at the boot-level, adds a folder icon to the screen labeled with the name of the file system (as stored in the device directory). The user is then able to click on the icon and thus to display the newly mounted file system.

5.2.1.8 File space allocations

The direct-access nature of disks allows flexibility in the implementation of files. In almost every case, many files will be stored on the same disk. The main problem is how to allocate space to these files so that disk space is utilized effectively and files can be accessed quickly. There are three major methods of allocating disk space:

(a) Contiguous space allocation

(b) Linked allocation

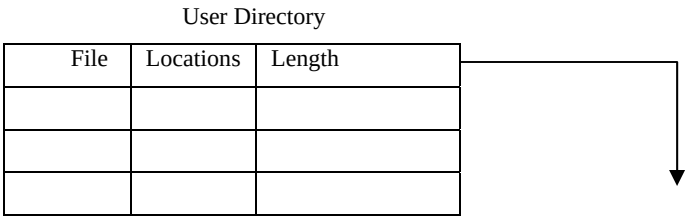
(c) Indexed allocation

Each method has its advantages and disadvantages. Accordingly some systems support all three. More common system will use one particular method for all files.

5.2.1.8.1 Contiguous space Allocation

The simplest scheme is contiguous allocation. The logical blocks of a file are stored in a partition of contiguous physical blocks. Disk addresses define a linear ordering on the disk. With this ordering, assuming that only one job is accessing the disk, accessing block b+1 after block b normally requites no head movement. When head movement is needed (from the last sector of one cylinder to the first sector of the next cylinder), it is only one-track movement. Thus, the number of disk seeks required for accessing contiguously allocated files is minimal. The disk address and length (in block units) of the first block define contiguous allocation of the file. If the file is n blocks long, and starts at location b, then it occupies block b, b+1, b+2, ..., b+n-1. The directory entry for each file indicates the address of the starting block and the length of the area allocated for this file. Accessing a file that has been allocated contiguously is easy. For sequential access, the file system remembers the disk address of the last block referenced and, when necessary, reads the next block. For direct access to block i of a file that starts at block b, we can immediately access block b+i. So contiguous space allocation easily supports both sequential and direct access.

The major problem with contiguous allocation is locating the space for a new file. The contiguous disk space-allocation problem can be seen to be particular application of the general dynamic storage-allocation problem, which is how to satisfy a request of size n from a list of free holes. First-fit (This strategy allocates the first available space that is big enough to accommodate file. Search may start at beginning of set of holes or where previous first-fit ended. Searching stops as soon as it finds a free hole that is large enough) and best-fit (This strategy allocates the smallest hole that is big enough to accommodate file. Entire list ordered by size is searched and matching smallest left over hole is chosen) are the most common strategies used to select a free hole from the set of available holes. Simulations have shown that both first-fit and best-fit are more efficient than worst-fit (This strategy allocates the largest hole. Entire list is searched. It chooses largest left over hole) in terms of both time and storage utilization. Neither first-fit nor best-fit is clearly best in terms of storage utilization, but first-fit is generally faster.



These algorithms suffer from the problem of external fragmentation i.e. the tendency to develop a large number of small holes. As files are allocated and deleted, the free disk space is broken into little pieces. External fragmentation exists whenever free space is broken into chunks. It becomes a problem when the largest contiguous chunk is insufficient for a request; storage is fragmented into a number of holes, no one of which is large enough to store the data. Depending on the total amount of disk storage and the average file size, external fragmentation may be either a minor or a major problem.

Some older microcomputer systems used contiguous allocation on floppy disks. To prevent loss of significant amounts of disk space to external fragmentation, the user had to run a repacking routine that copied the entire file system onto another floppy disk or onto a tape. The original floppy disk was then freed completely, creating one large contiguous free space. The routine then copied the files back onto the floppy disk by allocating contiguous space from this one large hole.

The scheme effectively compacts all free space into one contiguous space, solving the fragmentation problem. The cost of this compaction is time. The time cost is particularly severe for large hard disks that use contiguous allocation, where compacting all the space may take hours and may be necessary on a weekly basis. During this down time, normal

system operation generally cannot be permitted, so such compaction is avoided at all costs on production machines.

This is not all, there are other problems with contiguous allocation. A major problem is determining how much space is needed for a file. When the file is created, the total amount of space it will need must be found and allocated. How does the creator (program or person) know the size of the file to be created? In some cases, this determination may be fairly simple (copying an existing file, for example); in general, however, the size of an output file may be difficult to estimate.

If too little space is allocated to a file, it may be found that file cannot be extended. Especially with only a best-fit allocation strategy, the space on both sides of the file may be in use. Hence, we cannot make the file larger in space. Two possibilities then exist. First, the user program can be terminated, with an appropriate error message. The user must then allocate more space and run the program again. These repeated runs may prove costly. To prevent them, the user will normally overestimate the amount of space needed, resulting in considerable wasted space.

The other possibility is to find a larger hole, to copy the contents of the file to the new space and release the previous space. This series of actions may be repeated as long as space exists, although it can also be time-consuming. Notice, however, that in this case the user never needs to be informed explicitly about what is happening; the system continues despite the problem, although more and more slowly.

Even if the total amount of space needed for a file is known in advance, pre-allocation may be inefficient. A file that grows slowly over a long period (months or years) must be allocated enough space for its final size, even though much of that space may be unused for a long time. The file, therefore, has a large amount of internal fragmentation.

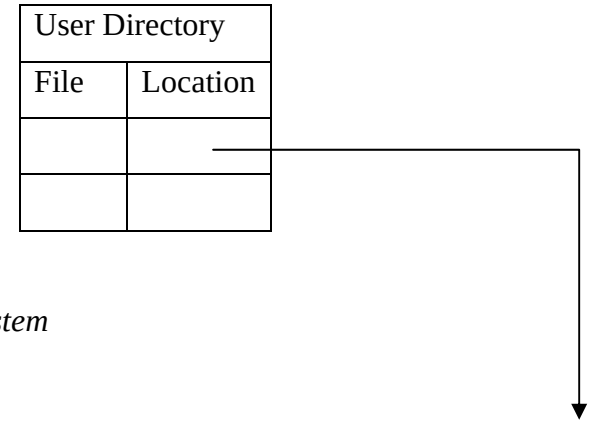
To avoid several of these drawbacks, some operating systems use a modified contiguous allocation scheme, in which a contiguous chunk of space is allocated initially, and then, when that amount is not large enough, another chunk of contiguous space, called an extent, is added to the initial allocation. The location of a file's blocks is then recorded as a location and a block count, plus a link to the first block of the next extent. On some systems, the owner of the file can set the extent size, but this setting results in inefficiencies if the owner is incorrect. Internal fragmentation can still be a problem if the extents are too large, and

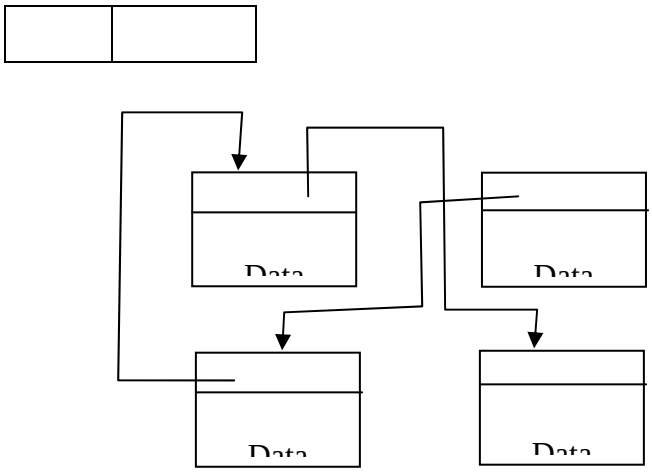
external fragmentation can be a problem as extents of varying sizes are allocated and de-allocated in turn.

5.2.1.8.2 Linked Allocation

In linked allocation, file is not stored on a contiguous set of blocks, rather the physical blocks in which a file is stored may be scattered throughout the secondary storage devices. Linked allocation solves all problems of contiguous allocation. With linked allocation, each file is a linked list of disk blocks; the disk blocks may be scattered anywhere on the disk. The directory contains a pointer to the first and last blocks of the file. For example, a file of five blocks might start at block 9, continue at block 16, then block 1, block 10, and finally block 25. Each block contains a pointer to the next block. These pointers are not made available to the user. Thus, if each block is 512 bytes, and a disk address (the pointer) requires 4 bytes, then the user sees blocks of 508 bytes. To create anew file, we simply create a new entry in the directory. With linked allocation, each directory entry has a pointer to the first disk block of the file. This pointer is initialised to nil (the end-of-list pointer value) to signify an empty file. The size field is also set to 0. A write to the file causes a free block to be found via the free-space management system, and this new block is then written to, and is linked to the end of the file. To read a file, we simply read blocks by following the pointers from block to block.

There is no external fragmentation with linked allocation, and any free block on the free-space list can be used to satisfy a request. Notice also that there is no need to declare the size of a file when a file is created. A file can continue to grow as long as there are free blocks. Consequently, it is never necessary to compact disk space. Linked allocation suffers from some disadvantages, however. The major problem is that it can be used effectively for only sequential-access files. To find the i^{th} block of a file, we must start at the beginning of that file, and follow the pointers until we get to the i^{th} block. Each access to a pointer requires a disk read, and sometimes a disk seek also. Consequently, it is inefficient to support a direct-access capability for linked allocation files.





Space required for the pointers is another disadvantage to linked allocation. If a pointer requires 4 bytes out of a 512-byte block, then ((4 / 512) * 100 = 0.78) percent of the disk is being used for pointers, rather than for information. Each file requires slightly more space than it otherwise would. The usual solution to this problem is to collect blocks into multiples, called clusters, and to allocate the clusters rather than blocks. For instance, the file system may define a cluster as 4 blocks, and operate on the disk in only cluster units. Pointers then use a much smaller percentage of the file's disk space. This method allows the logical-to-physical block mapping to remain simple, but improves disk throughput (fewer disk head-seeks) and decreases the space needed for block allocation and free-list management. The cost of this approach is an increase in internal fragmentation, because more space is wasted if a cluster is partially fully than when a block is partially full. Clusters can be used to improve the disk access time for many other algorithms, so they are used in most operating systems.

Yet another problem is reliability. Since the files are linked together by pointers scattered all over the disk, consider what would happen if a pointer were lost or damaged. A bug in the operating- system software or a disk hardware failure might result in picking up the wrong pointer. This error could result in linking into the free-space list or into another file. Partial solutions are to use doubly linked lists or, to store the file name and relative block number in each block; however, these schemes require even more overhead for each file.

An important variation on the linked allocation method is the use of a file-allocation table (FAT). This simple but efficient method of disk-space allocation is used by the MS-DOS and OS/2 operating systems. A section of disk at the beginning of each partition is reserved to contain the table. The table has one entry for each disk block, and is indexed by block

number. The FAT is used much as is a linked list. The directory entry contains the block number of the first block of the file. The table entry indexed by that block number then contains the block number of the next block in the file. This chain continues until the last block, which has a special end-of-file value as the table entry. Unused blocks are indicated by a 0 table value. Allocating a new block to a file is a simple matter of finding the first 0-valued table entry, and replacing the previous end-of-file value.

Note that the FAT allocation scheme can result in a significant number of disk head seeks, unless the FAT is cached. The disk head must move to the start of the partition to read the FAT and find the location of the block in question, then move to the location of the block itself. In the worst case, both moves occur for each of the blocks. A benefit is that random access time is improved, because the disk head can find the location of any block by reading the information in the FAT.

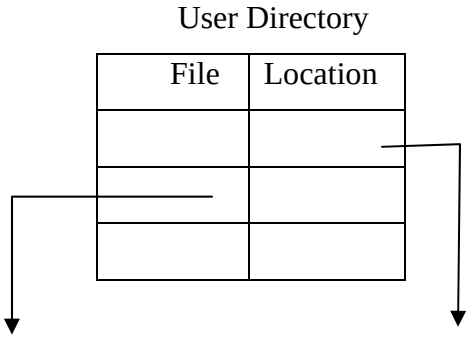
5.2.1.8.3 Indexed Allocation

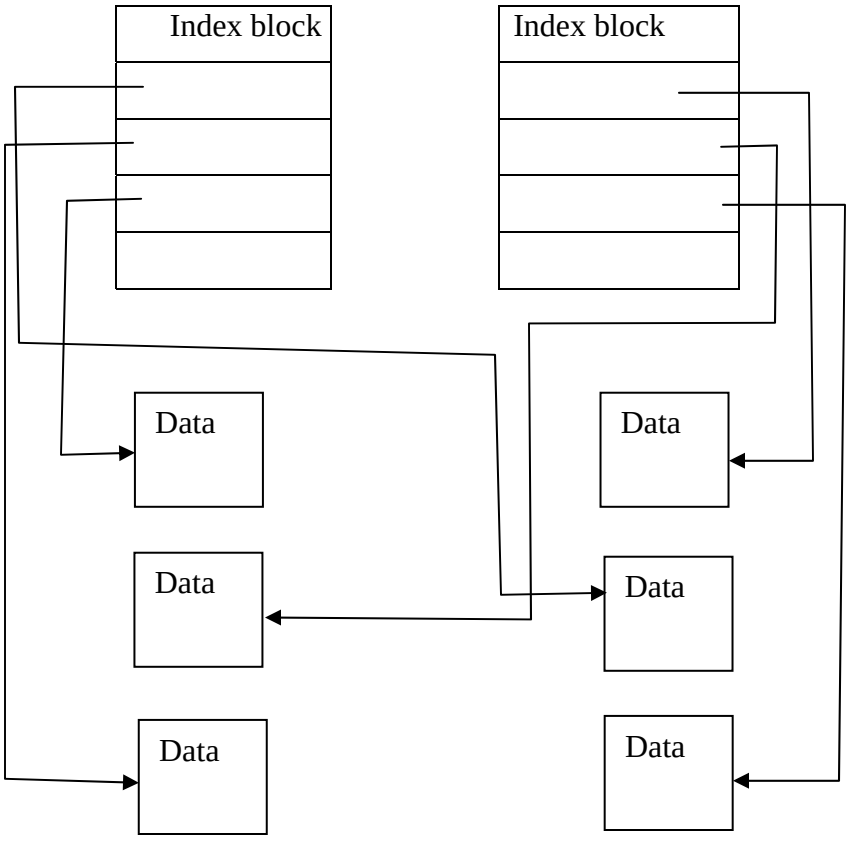
Although linked allocation solves the external-fragmentation and size-declaration problems of contiguous allocation. However, in the absence of a FAT, linked allocation cannot support efficient direct access, since the pointers to the blocks are scattered with the blocks themselves all over the disk and need to be retrieved in order. Indexed allocation solves this problem by bringing all the pointers together into one location, called the index block. Indexed allocation is a variant of linked allocation.

Each file has its own index block, which is an array of disk-block addresses. The i^{th} entry in the index block points to the i^{th} block of the file. The directory contains the address of the index block (See following figure). To read the i^{th} block, we use the pointer in the i^{th} index-block entry to find and read the desired block.

When the file is created, all pointers in the index block are set to nil. When the i^{th} block is first written, a block is obtained from the free-space manager, and its address is put in the i^{th} index-block entry.

Indexed allocation supports direct access, without suffering from external fragmentation, because any free block on the disk may satisfy a request for more space.





Indexed allocation of disk space

Indexed allocation does suffer from wasted space. The pointer overhead of the index block is generally greater than the pointer overhead of linked allocation.

Note that indexed allocation schemes suffer from some of the same performance problems, as does linked allocation. Specifically, the index blocks can be cached in memory, but the data blocks may be spread all over a partition.

5.2.1.8.4 Performance

To evaluate the performance of allocation methods, two important criteria are storage efficiency and data-block access times. Both are important criteria in selecting the proper method or methods for an operating system to implement.

One difficulty in comparing in performance of the various systems is determining how the systems will be used – in a sequential access manner or random access. A system with mostly sequential access should use a method different from that for a system with mostly random access. For any type of access, contiguous allocation requires only one access to get

a disk block. Since we can easily keep the initial address of the file in memory, we can calculate immediately the disk address of the i^{th} block (or the next block) and read it directly. For linked allocation, we can also keep the address of the next block in memory and read it directly. This method is fine for sequential access; for direct access, however, an access to the i^{th} block might require i disk reads. This problem indicates why linked allocation should not be used for an application requiring direct access.

As a result, some systems support direct-access files by using contiguous allocation and sequential access by linked allocation. For these systems, the type of access to be made must be declared when the file is created. A file created for sequential access will be linked and cannot be used for direct access. A file created for direct access will be contiguous and can support both direct access and sequential access, but its maximum length must be declared with it.

5.2.1.9 File attributes

Attributes are properties of a file. The operating system treats a file according to its attributes. Following are a few common attributes of a file:

- H for hidden
- A for archive
- D for directory
- X for executable
- R for read only

These attributes can be used in combination also.

5.2.2 ACCESS METHODS

Files store information, which is when required, may be read into the main memory. There are several different ways in which the data stored in a file may be accessed for reading and writing. The operating system is responsible for supporting these file access methods. The fundamental methods for accessing information in the file are (a) sequential access: in it information in the file must be accessed in the order it is stored in the file, (b) direct access, and (c) index sequential access.

5.2.2.1 Sequential access

A sequential file is the most primitive of all file structures. It has no directory and no linking pointers. The records are generally organized in a specific sequence according to the key

field. In other words, a particular attribute is chosen whose value will determine the order of the records. Access proceeds sequentially from start to finish. Operations to read or write the file need not specify the logical location within the file, because operating system maintains a file pointer that determines the location of the next access. Sometimes when the attribute value is constant for a large number of records a second key is chosen to give an order when the first key fails to discriminate. Use of sequential file requires data to be sorted in a desired sequence according to the key field before storing or processing them.

Its main advantages are:

- It is easy to implement
- It provides fast access to the next record if the records are to be accessed using lexicographic order.

Its disadvantages are:

- It is difficult to update and insertion of a new record may require moving a large proportion of the file
- Random access is extremely slow.

Sometimes a file is considered to be sequentially organised despite the fact that it is not ordered according to any key. Perhaps the date of acquisition is considered to be the key value, the newest entries are added to the end of the file and therefore pose no difficulty to updating. Sequential files are advisable if the applications are sequential by nature.

5.2.2.2 Index-sequential

An index-sequential file each record is supposed to have a unique key and the set of records may be ordered sequentially by a key. An index is maintained to determine the location of a record from its key value. Each key value appears in the index with the associated address of its record. To access a record with key k, the index entry containing k is found by searching the index and the disk address mentioned in the entry is used to access the record.

In the following figure an employee file is illustrated where records are arranged in ascending order according to the employee #.

Track #	
1	1 2 5 8 16 20 25 30 32 36
2	38 40 41 43 44 45 50 52
3	53 57 59 60 62 64 67 70

A track index is maintained as shown in the following figure to speed up the search:

Track	Low	High
1	1	36
2	38	52
3	53	70

For example, to locate the record of employee # 41, index is searched. It is evident from the index that the record of employee #41 will be on track no.2 because it has the lowest key value 48 and highest key value 52.

In the literature an index-sequential file is usually thought of as a sequential file with a hierarchy of indices. For example, there might be three levels of indexing: track, cylinder and master. Each entry in the track index will contain enough information to locate the start of the track, and the key of the last record in the track, which is also normally the highest value on that track. There is a track index for each cylinder. Each entry in the cylinder index gives the last record on each cylinder and the address of the track index for that cylinder. If the cylinder index itself is stored on tracks, then the master index will give the highest key referenced for each track of the cylinder index and the starting address of that track. No mention has been made of the possibility of overflow during an updating process. Normally provision is made in the directory to administer an overflow area. This of course increases the number of book-keeping entries in each entry of the index.

5.2.2.3 Direct access

In direct access file organization, any records can be accessed irrespective of the current position in the file. Direct access files are created on direct access storage devices. Whenever a record is to be inserted, its key value is mapped into an address using a hashing function. On that address record is stored. The advantage of direct access file organization is realized when the records are to be accessed

randomly (not sequentially). Otherwise this organization has a number of limitations such as (a) poor utilization of the I/O medium and (b) Time consumption during record address calculation.

5.3 Key words

Contiguous space Allocation: The logical blocks pf a file are stored in a partition of contiguous physical blocks.

Linked allocation: In it each file is a linked list of disk blocks; the disk blocks may be scattered anywhere on the disk. The directory contains a pointer to the first and last blocks of the file.

Indexed Allocation: In Indexed allocation all the pointers together are stored into one location, called the index block. Each file has its own index block, which is an array of disk-block addresses.

Sequential access: In it information in the file must be accessed in the order it is stored in the file.

Index-sequential: In index-sequential file each record is supposed to have a unique key and the set of records may be ordered sequentially by a key. An index is maintained to determine the location of a record from its key value.

Direct access: In direct access file organization, records can be accessed randomly. The key value of the record is mapped into an address using a hashing function. On that address record is stored.

5.4 SUMMARY

The file system resides permanently on secondary storage, which has the main requirement that it must be able to hold a large amount of data, permanently.

The various files can be allocated space on the disk in three ways: through contagious, linked or indexed allocation. Contagious allocation can suffer from external fragmentation. Direct-access is very inefficient with linked-allocation. Indexed allocation may require substantial overhead for its index block. There are many ways in which these algorithms can be optimised.

Free space allocation methods also influence the efficiency of the use of disk space, the performance of the file system and the reliability of secondary storage.

5.5 SELF-ASSESSMENT QUESTIONS (SAQ)

1. What do you understand by a file? What is a file system?
2. What are the different modes to share a file?
3. What are the different methods to access the information from a file? Discuss their advantages and disadvantages.
4. What are the advantages of indexed allocation over linked allocation and contiguous space allocation? Explain.
5. Differentiate between first fit, best fit and worst fit storage allocation strategies.

5.6 SUGGESTED READINGS / REFERENCE MATERIAL

1. Operating System Concepts, 5th Edition, Silberschatz A., Galvin P.B., John Wiley and Sons.
2. Systems Programming and Operating Systems, 2nd Revised Edition, Dhamdhere D.M., Tata McGraw Hill Publishing Company Ltd., New Delhi.
3. Operating Systems, Madnick S.E., Donovan J.T., Tata McGraw Hill Publishing Company Ltd., New Delhi.
4. Operating Systems-A Modern Perspective, Gary Nutt, Pearson Education Asia, 2000.
5. Operating Systems, Harris J.A., Tata McGraw Hill Publishing Company Ltd., New Delhi, 2002.

Lesson number: 6	***Writer: Dr. Rakesh Kumar***
File System - II	***Vetter: Dr. Pradeep Bhatia***

6.0 Objectives

The objectives of this lesson are to make the students familiar with directory system and file protection mechanism. After studying this lesson students will become familiar with:

(a) Different types of directory structures.

(b) Different protection structures such as:

- Access Control Matrix
- Access Control Lists

6.1 Introduction

A file system provides the following facilities to its users:

(a) Directory structure and file naming facilities,

(b) Protection of files against illegal form of access,

(c) Static and dynamic sharing of files, and

(d) Reliable storage of files.

A file system helps the user in organizing the files through the use of directories. A directory may be defined as an object that contains the names of the file system objects. Entries in the directory determine the names associated with a file system object. A directory contains information about a group of files. A typical structure of a directory entry is as under:

File name – Locations Information – Protection Information – Flags

The presences of directories enable file system to support file sharing and protection. Sharing is simply a matter of permitting a user to access the files of other user stored in some other directory. Protection is implemented by permitting the owner of a file to specify which other users may access his files and in what manner. All these issues are discussed in detail in this lesson.

6.2 Presentation of Contents

6.2.1 Hierarchical Directory Systems

6.2.1.1 Directory Structure

6.2.1.2 The Logical Structure of a Directory

6.2.1.2.1 Single-level Directory

6.2.1 Hierarchical Directory Systems

Files are generally stored on secondary storage devices. Numerous files are to be stored on storage of giga-byte capacity. To handle such a huge size of data, there is a need to properly organize the files. The organization, usually, done in two parts. In the first part, a file system may incorporate the notion of a partition, which determines on which device a file will be stored. The file system is broken into partitions, also known as minidisks or volumes. Typically, a disk contains at least one partition, which is a low-level structure in which files and directories reside. Sometimes, there may be more than one partition on a disk, each partition acting as a virtual disk. The users do not have to concern themselves with the translating the physical address; the system does the required job.

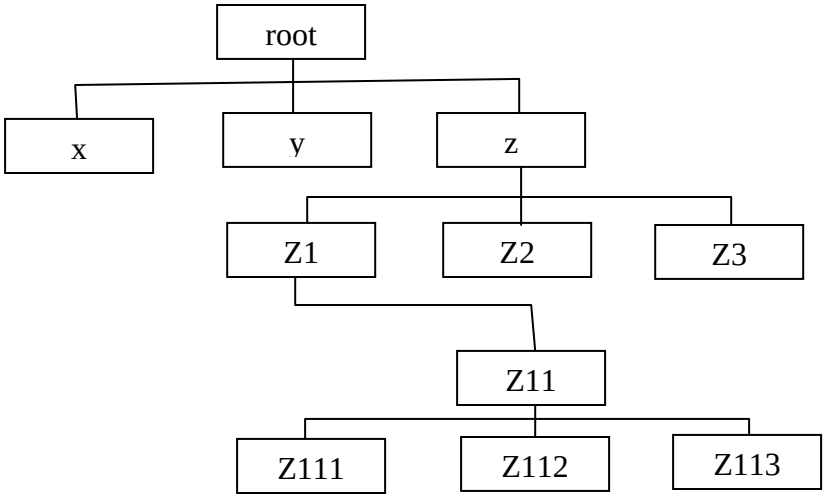


Figure 1 Directory Hierarchy

Partitions contain information about itself in a file called partition table. It also contains information about files and directories on it. Typical file information is name, size, type, location etc. The entries are kept in a device directory or volume table of contents (VTOC).

Directories may be created within another directory. Directories have parent-child relationship as shown in the Figure 1.

6.2.1.1 Directory Structure

The file systems of computers can be extensive. Some systems store thousands of files on hundreds of gigabytes of disk. To manage all these data, we need to organize them. This organization is usually done in two parts; first, the file system is broken into in the IBM world or volumes in the PC and Macintosh arenas. Sometimes, partitions are used to provide several separate areas within one disk, each treated as a separate storage device, whereas other systems allow partitions to be larger than a disk to group disks into one logical structure. In this way, the user needs to be concerned with only the logical directory and file structure, and can ignore completely the problems of physically allocating space for files. For this reason partitions can be thought of as virtual disks.

Second, each partition contains information about files within it. This information is kept in a device directory or volume table of contents. The device directory (more commonly known

simply as a "directory") records information such as name, location, size, and type for all files on that partition.

6.2.1.2 The Logical Structure of a Directory

6.2.1.2.1 Single-Level Directory

The simplest directory structure is the single-level tree. A single level tree system has only one directory. All files are contained in the same directory, which is easy to support and understand. Names in that directory refer to files or other non-directory objects. Such a system is practical only on systems with very limited numbers of files. The advantage of this structure is its simplicity only. But it has got many limitations:

Limitations:

A single-level directory is manageable when the number of files is small. When the number of files increases or when there is more than one user, it suffers from the following problems:

(a) Since all files are stored in the same directory, the name given to each file should be unique. If there are two users and they give the same name to their file, then there is a problem.

(b) Even with a single user, as the number of files increase, it becomes difficult to remember the names of all the files, so as to create only files with unique names. It is not uncommon for a user to have hundreds of files on one computer system and an equal number of additional files on another system. In such an environment, keeping track of so many files is a daunting task.

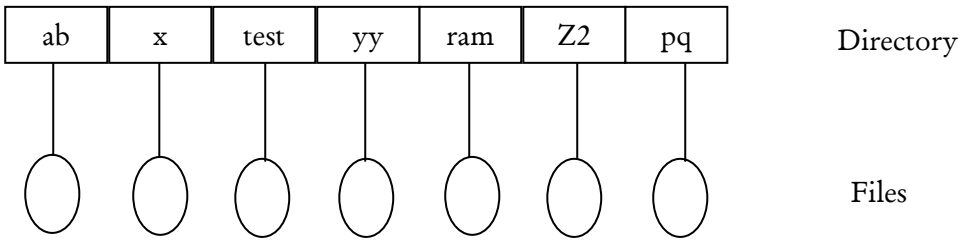


Figure 2 Single-level Directory

6.2.1.2.2 Two-Level Directory

The disadvantage of a single-level directory is confusion of file names. The standard solution is to create a separate directory for each user. In a two level system, only the root level directory may contain names of directories and all other directories refer only to non-

directory objects. In the two-level directory structure, each user has his/her own user file directory (UFD). Each UFD has a similar structure, but lists only the files of a single user. When a user starts or a user logs in, the system's master file directory is searched. The master file directory is indexed by user name or account.

When in a UFD a user refers to a particular file, only his own UFD is searched. Thus, different users may have files with the same name, as long as till the filenames within each UFD are unique. To create a file for a user, the operating system searches only that user's UFD to ascertain whether another file of that name exists. To delete a file, the operating system confines its search to the local UFD; thus, it cannot accidentally delete another user's file that has the same name.

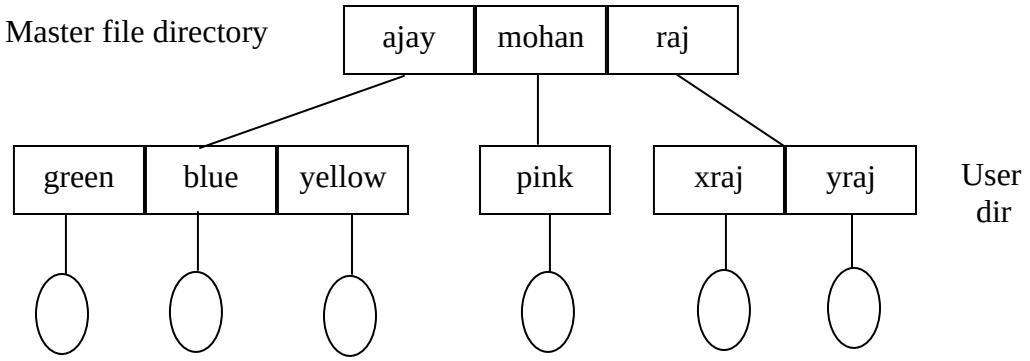


Figure 3 Two-Level Directory

The user directories themselves must be created and deleted as necessary. A special system program is run with the appropriate user name and account information. The program creates a new user file directory and adds an entry for it to the master file directory. The execution of this program might be restricted to system administrators.

The two-level directory structure solves the name-collision problem, but it still has problems. This structure effectively isolates one user from another. This isolation is an advantage when the users are completely independent, but is a disadvantage when the users co-operate on some task and to access one user's account by other users is not allowed. If access is to be permitted, one user must have the ability to name a file in another user's directory.

A two-level directory can be thought of as a tree, or at least an inverted tree. The root of the tree is the master file directory. Its direct descendants are the UFDs. The descendants of the user file directories are the files themselves. Thus, a user name and a file name define a path

name. Every file in the system has a path name. To name a file uniquely, user must know the path name of the file desired.

For example, if user A wishes to access her own test file named *test*, he can simply refer to *test*. To access the test file of user B (with directory-entry name *userb*), however, he might have to refer to */userb/test*. Every system has its own syntax for naming files in directories other than the user's own.

There is additional syntax to specify the partition of a file. For instance, in MS-DOS a letter followed by a colon specifies a partition. Thus, file specification might be "C:\userb\bs.test". Some systems go even further and separate the partition, directory name, and file name parts of the specification. For instance, in VMS, the file "login.com" might be specified as:

"u:[sst.deck.1]login.com;"

where "u" is the name of the partition, "sst" is the name of the directory, "deck" is the name of subdirectory, and "1", is the version number. Other systems simply treat the partition name as part of the directory name. The first name given is that of the partition, and the rest is the directory and file. For instance, "/u/pbg/test" might specify partition "u", directory "pbg", and file "test". A special case of this situation occurs in regard to the system files. Those programs provided as parts of the system (loaders, assemblers, compilers, utility routines, libraries, and so on) are generally defined as files. When the appropriate commands are given to the operating system, these files are read by the loader and are executed. Many command interpreters act by simply treating the command as the name of a file to load and execute. As the directory system is defined presently, this file name would be searched for in the current user file directory .One solution would be to copy the system files into each user file directory. However, copying all the system files would be enormously wasteful of space. The standard solution is to complicate the search procedure slightly. A special user directory is defined to contain the system files.

Whenever a file name is given to be loaded, the operating system first searches the local user file directory. If the file is found, it is used. If it is not found, the system automatically searches the special user directory that contains the system files. The sequence of directories searched when a file is named is called the search path. This idea can be extended, such that the search path contains an unlimited list of directories to search when a command name is given. This method is used in UNlX and MS-DOS.

6.2.1.2.3 Tree-Structured Directories

A tree system allows growth of the tree beyond the second level. Any directory may contain names of additional directories as well as non-directory objects. This generalization allows users to create their own sub-directories and to organize their files accordingly. The MS-DOS system, for instance, is structured as a tree. In fact, a tree is the most common directory structure. The tree has a root directory. Every file in the system has a unique path name. A path name is the path from the root, through all the subdirectories, to a specified file.

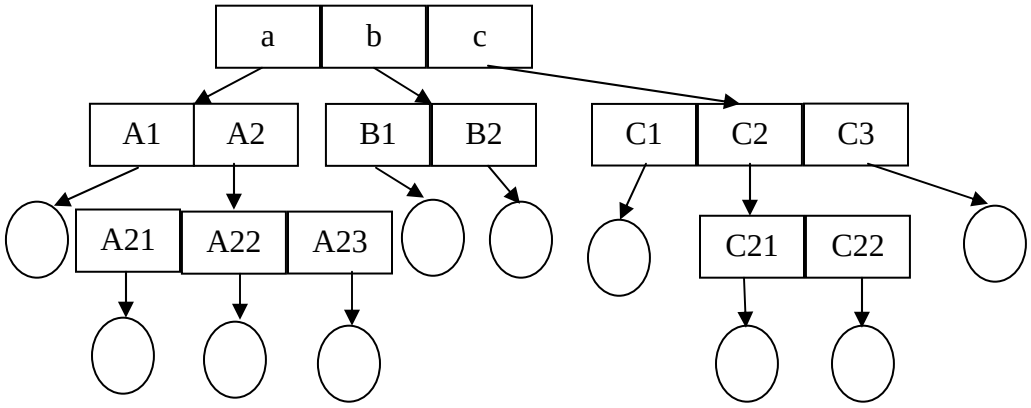


Figure 4 Tree-Structured Directories

A directory (or subdirectory) contains a set of files or subdirectories. A directory is simply another file but it is treated in a special way. All directories have the same internal format, one bit in each directory entry defines the entry as a file (0) or as a subdirectory (1) Special system calls are used to create and delete directories.

In normal use, each user has a current directory .The current directory should contain most of the files that are of current interest to the user. When reference is made to a file, the current directory is searched. If a file is needed that is not in the current directory, then the user must either specify a path name or change the current directory to be the directory holding that file. To change the current directory to a different directory, a system call is provided that takes a directory name as a parameter and uses it to redefine the current directory.

Thus, the user can change his current directory whenever he desires. From one change directory system call to the next, all open system calls search the current directory for the specified file.

The initial current directory of a user is designated when the user job starts or the user logs in. The operating system searches the accounting file (or ask) other predefined location to

find an entry for this user (for accounting). In the accounting file is a pointer to (or the name of) the user's initial directory. This pointer is copied to a local variable for this user, which specifies the user's initial current directory.

Path names can be of two types: absolute path names or relative path names.

(a) **Absolute path:** An absolute path name begins at the root and follows a path down to the desired file, giving the directory names on the path. An absolute path name is an unambiguous way of referring to a file. Thus identically named files created by different users differ in their absolute path names.

(b) **Relative path:** A relative path name defines a path from the current directory.

Allowing the user to define his own subdirectories permits him to impose a structure on his files. This structure might result in separate directories for files associated with different topics (for example, a subdirectory was created to hold the text of this book or different forms of information for example, the directory programs may contain source programs; the directory bin may store all the binary files. An interesting policy decision in a tree-structured directory structure is how to handle the deletion of a directory. If a directory is empty, its entry in its containing directory can simply be deleted. But if the directory to be deleted is not empty, containing files and subdirectories then one of two approaches can be taken. As in MS-DOS, if we want to delete a directory then first of all we have to empty it i.e. delete its contents and If there are any subdirectories, the procedure must be applied recursively to them, so that they can be deleted also. But this approach may be time consuming.

An alternative approach, such as that taken by the UNIX rm command, to provide the option that, when a request is made to delete a directory, and that directory's files and subdirectories are also to be deleted. Note that either approach is fairly easy to implement; the choice is one of policy. The latter policy is more convenient, but more dangerous, because an entire director structure may be removed with one command. If that command was issued in error, a large number of files and directories would need to be restored from backup tapes.

With a tree-structured directory system, users can access, in addition their files, the files of other users. For example, user B can access files of user A by specifying their path names. User B can specify either an absolute or relative path name. Alternatively, user B could change her current directory be user A's directory, and access the files by their file names. Some systems also allow users to define their own search paths. In this case, user B could

define her search path to be (1) her local directory, (2) the system file directory, and user A's directory, in that order. As long as the name of a file of user A did not conflict with the name of a local file or system file, it could be referred to simply by its name.

A path to a file in a tree-structured directory can be longer than that in a two-level directory. To allow users to access programs without having to remember these long paths, the Macintosh operating system automates the search for executable programs. It maintains a file called the "Desktop File", containing the name and location of all executable programs it has seen. Where a new hard disk or floppy disk is added to the system, or the network accessed, the operating system traverses the directory structure, searching for executable programs on the device and recording the pertinent information. This mechanism supports the double-click execution functionality. A double-click on a file causes its creator attribute to be read, and the "Desktop File" to be searched for a match.

6.2.1.2.4 Acyclic-Graph Directories

Sharing of file is another important issue in deciding the directory structure. If more than one user are working on some common project. So the files associated with that project should be placed in a common directory that can be shared among a number of users.

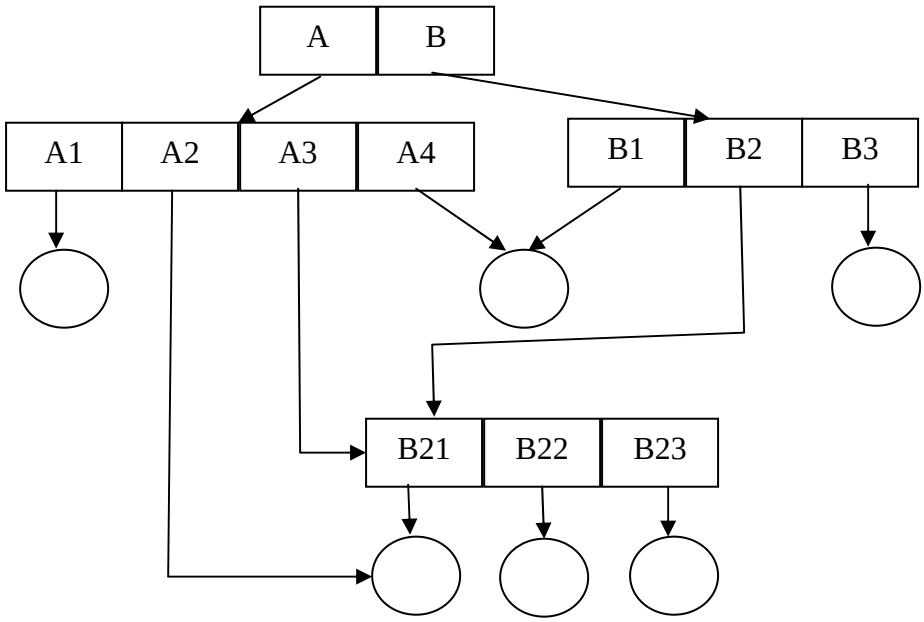


Figure 5 Acyclic-Graph Directories

The important characteristic of sharing is that if a user is making a change in a shared file then that is to be reflected to other user also. In this way a shared file is not the same as two copies of the file. With two copies, each programmer can view the copy rather than the original, but if one programmer changes the file, the changes will not appear in the other's copy. With a shared file, there is only one actual file, so any changes made by the person would be immediately visible to the other.

This form of sharing is particularly important for shared subdirectories; a new file created by one person will automatically appear in all the shared subdirectories. File sharing is facilitated by acyclic graph structure. The tree structure doesn't permit the sharing of files.

In a situation where several people are working as a team, all the files to be shared may be put together into one directory. The user file directories of all the team members would each contain this directory of shared files as a subdirectory. Even when there is a single user, his file organization may require that some files be put into several different subdirectories. For example, a program written for a particular project should be both in the directory of all programs and in the directory for that project.

Shared files and subdirectories can be implemented in several ways. A common approach used in UNIX systems, is to create a new directory entry called a link. A link is a pointer to another file or subdirectory. For example, a link may be implemented as an absolute or relative path name. When a reference to a file is made, we search the directory. The directory entry is marked as a link and the name of the real file (or directory) is given. We resolve the link by using the path name to locate the real file. Links are easily identified by their format in the directory entry (or by their having a special type on systems that support types), and are effectively named indirect pointers. The operating system ignores these links when traversing directory trees to preserve the acyclic structure of the system.

The other approach to implementing shared files is simply to duplicate all information about them in both sharing directories. Thus, both entries are identical and equal. A link is clearly different from the original directory entry; thus, the two are not equal. Duplicate directory entries, however, make the original and the copy indistinguishable. A major problem with duplicate directory entries is maintaining consistency if the file is modified. An acyclic-graph directory structure is more flexible than is a simple tree structure, but is also more

complex. Several problems must be considered carefully. Notice that a file may now have multiple absolute path names. Consequently, distinct file names may refer to the same file. This situation is similar to the aliasing problem for programming languages. If we are trying to traverse the entire file system this problem becomes significant, since we do not want to traverse shared structures more than once.

Another problem involves deletion. When can the space allocated to a shared file be de-allocated and reused? One possibility is to remove the file whenever anyone deletes it, but this action may leave dangling pointers to the now non-existent file. Worse, if the remaining file pointers contain actual disk addresses, and the space is subsequently reused for other files, these dangling pointers may point into the middle of other files.

In a system where sharing is implemented by symbolic links, this situation is somewhat easier to handle. The deletion of a link does not need to affect the original file; only the link is removed. If the file entry itself is deleted, the space for the file is de-allocated, leaving the links dangling. We can search for these links and remove them also, but unless a list of the associated link is kept with each file, this search can be expensive. Alternatively, we can leave the links until an attempt is made to use them. At that time, we can determine that the file of the name given by the link does not exist, and can fail to resolve the link name; the access is treated just like any other illegal file name. (In this case, the system designer should consider carefully what to do when a file is deleted and another file of the same name is created, before a symbolic link to the original file is used.) In the case of UNIX, symbolic links are left when a file is deleted, and it is up to the user to realize that the original file is gone or has been replaced.

Another approach to deletion is to preserve the file until all references to it are deleted. To implement this approach, we must have some mechanism for determining that the last reference to the file has been deleted. We could keep a list of all references to a file (directory entries or symbolic links). When a link or a copy of the directory entry is established, a new entry is added to the file-reference list. When a link or directory entry is deleted, we remove its entry on the list. The file is deleted when its file-reference list is empty. The trouble with this approach is the variable and potentially large size of the file-reference list.

However, we really do not need to keep the entire list -we need to keep only a count of the number of references. So a reference count is maintained with shared file, whenever a reference is made to it, it is incremented by one. On deleting a link, the reference count is decremented by one, when it becomes zero the file can be deleted. The UNIX operating system uses this approach for non-symbolic links, or hard links, keeping a reference count in the file information block or inode. By effectively prohibiting multiple references to directories, we maintain an acyclic-graph structure.

To avoid these problems, some systems do not allow shared directories links. For example, in MS-DOS, the directory structure is a tree structure, rather than an acyclic graph, thereby avoiding the problems associated with file deletion in an acyclic-graph directory structure.

6.2.1.2.5 General Graph Directory

One serious problem with using an acyclic graph structure is ensuring that there are no cycles. If we start with a two-level directory and allow users to create subdirectories, a tree-structured directory results. It should be fairly easy to see that simply adding new files and subdirectories to existing tree structure preserves the tree-structured nature. However, when we add links to an existing tree-structured directory, the tree structure is destroyed, resulting in a simple graph structure.

The primary advantage of an acyclic graph is the relative simplicity of the algorithms to traverse file in the graph and to determine when there are no more references to a file. We want to avoid file traversing shared sections of an acyclic graph twice, mainly for performance reasons. If we have just searched a major shared subdirectory for a particular file, without finding that file, we want to avoid searching that subdirectory again; the second search would be a waste of time.

To improve the performance of the system we should avoid searching any component twice in the systems where cycles are permitted. If cycles are not identified by the algorithm then it can be trapped in an infinite loop. One solution is to arbitrarily limit the number of directories, which will be accessed during a search.

A similar problem exists when we are trying to determine when a file can be deleted. As with acyclic-graph directory structures, a value zero in the reference count means that there are no more references to the file or directory, and the file can be deleted. However, it is also possible, when cycles exist, that the reference count may be nonzero, even when it is no

longer possible to refer to a directory or file. This anomaly is due to the self-referencing (a cycle) in the directory structure. In this case, it is generally necessary to use a garbage collection scheme to determine when the last reference has been deleted and the disk space can be reallocated.

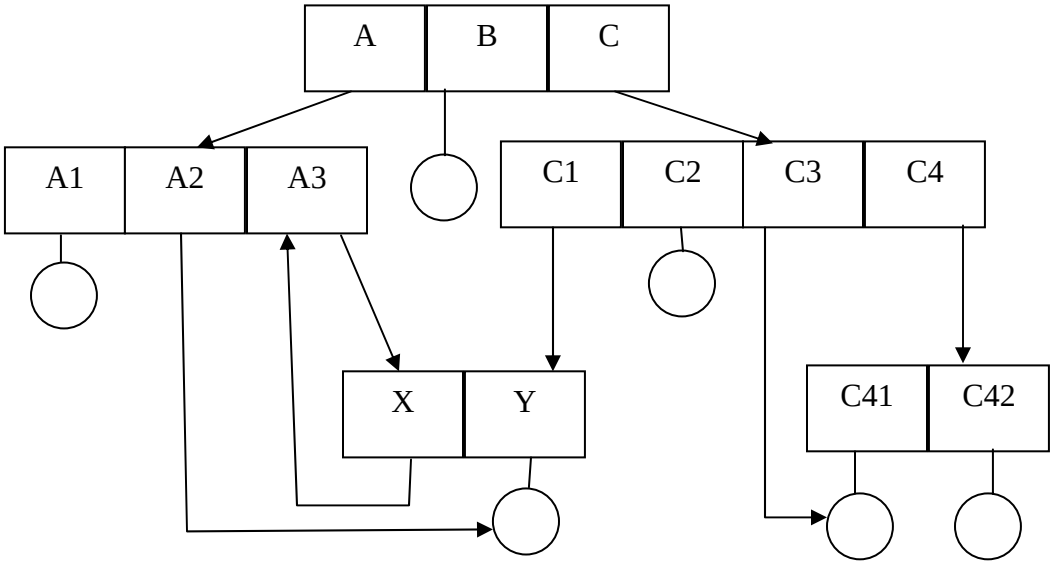


Figure 6 General Graph Directory

Garbage collection involves traversing the entire file system, marking everything that can be accessed. Then, a second pass collects everything that is not marked onto a list of free space. Garbage collection for a disk based file system, however, is extremely time-consuming and is thus seldom attempted. Garbage collection is necessary only because of possible cycles in the graph. Thus, an acyclic-graph structure is much easier to work with. The difficulty is to avoid cycles, as new links are added to the structure. There are algorithms to detect cycles in graphs. However, they are computationally expensive, especially when the graph is on disk storage. Generally, tree directory structures are more common than are acyclic-graph structures.

6.2.1.3 Directory Operations

The directory can be viewed as a symbol table that translates file names into their directory entries. If we take such a view, then it becomes apparent that the directory itself can be organized in many ways. The different operations that are to be carried out on directories are:

(a) To insert entries.

(b) To delete entries.

(c) To search for a named entry.

(d) To list all the entries in the directory.

When considering a particular directory structure, we need to keep in mind the operations that are to be performed on a directory:

- **Search for a directory:** We need to be able to search a directory structure to find the entry for a particular file. Since files have symbolic names and similar names may indicate a relationship between files, we may want to be able to find all files whose names match a particular pattern.
- **Create a directory:** New files need to be created and added to the directory.
- **Delete a directory:** When a file is no longer needed, we want to remove it from the directory.
- **List a directory:** We need to be able to list the files in a directory and the contents of the directory entry for each file in the list.
- **Rename a directory:** Because the name of a file represents its contents to its users, the name must be changeable when the contents or use of the file changes. Renaming a file may also allow its position within the directory structure to be changed.
- **Traverse the file system:** It is useful to be able to access every directory and every file within a directory structure. For reliability it is a good idea to save the contents and structure of the entire file system at regular intervals. This saving often consists of copying all files to magnetic tape. This technique provides a backup copy in case of system failure or if the file is simply no longer in use. In this case, the file can be copied to tape, and the disk space of that file released for reuse by another file.
- **Copying a directory:** A directory may be copied from one location to another.
- **Moving a directory:** A directory may be moved from one location to a new location with all its contents.

6.2.2 File Protection & Security

The security of the information is a major issue in file system. The files are to be protected from the physical damage as well as improper access. One way of ensuring the security is through backup. By maintaining the duplicate copy of the files, the reliability is improved. In many systems this is done automatically without human intervention. The backup of the files

is done at regular interval automatically. So if a copy of the file is accidentally destroyed, we have its backup copy.

There are a number of factors causing the damage to the file system such as:

(a) Hardware problems.
(b) Power failure
(c) Head crashes
(d) Dirt
(e) Temperature
(f) Bugs in the software

These things can result into the loss of contents of files. Protection can be provided in many ways. For a small single-user system, we might provide protection by physically removing the floppy disks and locking them in a desk drawer or file cabinet. In a multi-user system, however, other mechanisms are needed.

6.2.2.1 Types of Access

The need for protecting files is a direct result of the ability to access files. On systems that do not permit access to the files of other users, protection is not needed. Thus, one extreme would be to provide complete protection by prohibiting access. The other extreme is to provide free access with no protection. Both of these approaches are too extreme for general use. What is needed is the controlled access.

Protection mechanisms provide controlled access by limiting the types of file access that can be made. Access is permitted or denied depending on several factors, one of which is the type of access requested. Several different types of operations may be controlled:

- **Read** - Read information contained in the file.
- **Write** - Write new information into a file at any point or overwrite existing information in a file.
- **Execute** - Load the contents of a file into main memory and create a process to execute it.
- **Append** - Write new information at the end of the file.
- **Delete** - Delete the file and release its storage space for use in other files.

- **List** – Read the names contained in a directory.
- **Change access** - Change some user's access rights for some controlled operation.

Other operations, such as renaming, copying, or editing the file, may also be controlled. For many systems, however, these higher-level functions (such as copying) may be implemented by a system program that makes lower-level system calls. Protection is provided at only the lower level. For instance, copying a file may be implemented simply by a sequence of read requests. In this case, a user with read access can also cause the file to be copied, printed, and so on.

Many different protection mechanisms have been proposed. Each scheme has its advantages and disadvantages and must be selected as appropriate for intended application. A small computer system that is used by only a few members of a research group may not need the same types of protection as will a large corporate computer that is used for research, finance, and personnel iterations.

6.2.2.2 Protection structures

An access privilege is a right to make a specific form of access to a file. An access descriptor describes access privileges for a file. The common accesses privileges read, write, and execute are generally represented by r, w, and x descriptors. A user holds access privileges to one or more files and a file is accessible to one or more users. Access control information for a file is a collection of access descriptors for access privileges held by various users. Access control information can be organized in various forms such as Access Control Matrix, access Control Lists etc. which are discussed in the following section:

3.2.2.2.1 Access Control Matrix

Access control matrix (ACM) consists of rows and columns as shown in the following figure. Each row describes the access privileges held by a user. Each column describes the access control information for a file. Thus ACM (u_i, f_j) =a_{ij} implies that user u_i can access file f_j in accordance with access privileges a_{ij}.

Files→ Users ↓	f_1	f_2	f_3
u_1	{r}	(r, w}	{r, w, x}
u_2		{r}	{r, x}

u_3	{w}	{r, w, x}	{r}

Figure 7 Access Control Matrix

The important advantages of ACM are:

(a) Simplicity and efficiency of access.

(b) All information is stored in one structure.

But its main drawback is its size and sparseness. The size can be reduced by assigning access privileges to group of users rather than the individual users resulting in the reduction of number of rows. The solution of sparseness is the use of lists instead of matrix as discussed following.

3.2.2.2.2. Access control Lists and Groups

The most common approach to the protection problem is to make access dependent on the stems identity of the user. Various users may need different types of access to a file or directory. The most general scheme to implement identity-dependent access is to associate with each file and directory an access control list (ACL), specifying the user name and the types of access allowed for each user. Each element of the access control list is an access control pair (<user name>, <list of access privileges>).

When a user requests access to a particular file, the operating system checks the access list associated with that file. If that user is listed for the requested access, the access is allowed. Otherwise, a protection violation occurs, and the user job is denied access to the file.

The main problem with access lists is their length. It depends on the number of users and the number of access privileges defined in the system. Most file systems uses three kinds of access privileges: (a) Read - file can be read, (b) write – file can be modified and new data can be added, and (c) execute – permits the execution of the program. If we want to allow everyone to read a file, we must list all users with read access. This technique has two undesirable consequences:

(a) Constructing such a list may be a tedious and unrewarding task, especially if we do not know in advance the list of users in the system.

(b) The directory entry that previously was of fixed size needs now to be of variable size, resulting in space management being more complicated.

To reduce the size of protection information, users can be classified in some convenient manner and an access control pair can be specified for each class of user rather than for individual users. Now an access control list has only as many pairs as the number of user

classes. To condense the length of the access list, many systems recognize three classifications of users in connection with each file (e.g. in UNIX):

1. Owner - The user who created the file is the owner
2. Group - A set of users who are sharing the file and need similar access is a group or workgroup.
3. Universe - All other users in the system constitute the universe.

Note that, for this scheme to work properly, group membership must be controlled tightly. This control can be accomplished in a number of different ways. For example, in the UNIX system, groups can be created and modified by only the manager of the facility (or by any super-user). Thus, this control is achieved through human interaction. In the VMS system, with each file, an access list (also known as an access control list) may be associated, listing those users who can access the file. The owner of the file can create and modify this access lists are discussed above.

With this more limited protection classification, only three fields are needed to define protection. Each field is often a collection of bits, each of which either allows or prevents the access associated with it. For example, the UNIX system defines three fields of 3 bits each: rwx, where r controls read access, w controls write access, and x controls execution. A separate field is kept for the file owner, for the owner's group and for all other users. In this scheme, 9 bits per file are needed to record protection information.

3.2.2.2.3 Other Protection Approaches

Another approach to the protection problem is to associate a password with each file. Access to each file can be controlled by a password. If the passwords are chosen randomly and changed often, this scheme may be effective in limiting access to a file to only those users who know the password.

There are several disadvantages to this scheme.

(a) First, if we associate a separate password with each file, then the number of passwords that a user needs to remember may become large, making the scheme impractical.

(b) If only one password is used for all the files, then, once it is discovered, all files are accessible.

Some systems allow a user to associate a password with a subdirectory, rather than with an individual file, to deal with this problem. The IBM VM/CMS operating system allows three

passwords for a minidisk: one each for read, write, and multi write access. Second, commonly, only one password is associated with each file. Thus, protection is on an all-or-nothing basis. To provide protection on a more detailed level, we must use multiple passwords.

Limited file protection is also currently available on single user systems, such as MS-DOS and Macintosh operating system. These operating systems, when originally designed, essentially ignored dealing with the protection problem. However, since these systems are being placed on networks where file sharing and communication is necessary, protection mechanisms have to be retrofitted into the operating system. Note that it is almost always easier to design a feature into a new operating system than it is to add a feature to an existing one. Such updates are usually less effective and are not seamless.

We note that, in a multilevel directory structure, we need not only to protect individual files, but also to protect collections of files contained in a subdirectory, that is, we need to provide a mechanism for directory protection.

The directory operations that must be protected are somewhat different from the file operations. We want to control the creation and deletion of files in a directory. In addition, we probably want to control whether a user can determine the existence of a file in a directory. Sometimes, knowledge of the existence and name of a file may be significant in itself. Thus, listing the contents of a directory must be a protected operation. Therefore, if a path name refers to a file in a directory, the user must be allowed access to both the directory and the file. In systems where files may have numerous path names (such as acyclic or general graphs), a given user may have different access rights to a file, depending on the path name used.

3.3 Keywords

Directory: A directory may be defined as an object that contains the names of the file system objects.

Access Control Matrix: ACM is a matrix in which each row describes the access privileges held by a user and each column, access control information for a file.

Access Control List: It is a structure to implement identity-dependent access to each file and directory where each element of the ACL is an access control pair (<user name>, <list of access privileges>).

3.4 Summary

A file system helps the user in organizing the files through the use of directories that contains information about a group of files. A number of directory structures are used such as Single-level Directory Two-level Directory, Tree-structured Directories, Acyclic-Graph Directories, and General Graph Directory. Each approach has its merits and demerits. A number of operations are carried out on directories such as insertion, deletion, search, rename, traversal etc. So file system should facilitate these operations. Another important issue in file system is the protection of the information from physical damage and unauthorized access. To provide the access privileges to the files to different users two common mechanisms Access Control Matrix and Access Control Lists were discussed. ACM are characterized by their simplicity and efficiency but suffers from large size and sparseness. The problem of size is tackled by using the groups. If there are a number of blank entries in the ACM, then ACL can be a preferred solution.

6.5 Self-Assessment Questions (SAQ)

1. Define field, record, file, file sharing, and file protection.
2. What are the limitations of acyclic directory structure?
3. Which file operations are applicable to directories? Which are not?
4. How is a directory different from a file?
5. What are the different logical structures of the directory? Discuss their merits and demerits?
6. Discuss the advantages and disadvantages of Access Control Lists (ACL) and Access Control Matrix (ACM).
7. Discuss the advantages and disadvantages of two-level directory structure over single-level directory structure.

6.6 Suggested Readings / Reference Material

6. Operating System Concepts, 6th Edition, Silberschatz A., Galvin P.B., John Wiley & Sons.
7. Systems Programming & Operating Systems, 2nd Revised Edition, Dhamdhere D.M., Tata McGraw Hill Publishing Company Ltd., New Delhi.
8. Operating Systems, Madnick S.E., Donovan J.T., Tata McGraw Hill Publishing Company Ltd., New Delhi.

9. Operating Systems-A Modern Perspective, Gary Nutt, Pearson Education Asia, 2000.
10. Operating Systems, Har ris J.A., Tata McGraw Hill Publishing Company Ltd., New Delhi, 2002.

Lesson number: 7	***Writer: Dr. Rakesh Kumar***
Disk Scheduling	***Vetter: Dr. Pradeep Bhatia***

7.0 Objectives

The objective of this lesson is to make the students familiar with:

(a) The characteristics of the disk that affect the performance.
(b) A number of disk scheduling algorithms to improve the access time.
(c) Disk scheduling algorithm in fixed-head storage devices.
(d) RAID Technology

7.1 Introduction

We know that the processor and memory of the computer should be scheduled such that the system operates more efficiently. Another very important scheduler is the disk scheduler. The disk can be considered the one I/O device that is common to every computer. Most of the processing of computer system centers on the disk system. Disk provides the primary on-line storage of information, both programs and data. All the important programs of the system such as compiler, assemblers, loaders, editors, etc. are stored on the disk until loaded into memory. Hence it becomes all-important to properly manage the disk storage and it scheduling.

7.2 Presentation of contents

7.2.1 Storage device characteristics
- 7.2.1.1 Seek time
- 7.2.1.2 Latency time
- 7.2.1.3 Transfer time

7.2.2 Disk Scheduling

7.2.1 Storage device characteristics

Disk comes in many sizes and speeds, and information may be stored optically or magnetically. However, all disks share a number of important features. A disk is a flat circular object called a platter. Information may be stored on both sides of a platter (although some multiplatter disk packs do not use the top most or bottom most surfaces). Platter rotates around its own axis. The circular surface of the platter is coated with a magnetic material on which the information is stored. A read/write head is used to perform read/write operations on the disk. The read write head can move radially over the

magnetic surface. For each position of the head, the recorded information forms a circular track on the disk surface. Within a track information is written in blocks. The blocks may be of fixed size or variable size separated by block gaps. The variable length size block scheme is flexible but difficult to implement. Blocks can be separately read or written. The disk can access any information randomly using an address of the record of the form (track no, record no.). When the disk is in use a drive motor spins it at high speed. The read/write head positioned just above the recording surface stores the information magnetically on the surface. On floppy disks and hard disks, the media spins at a constant rate. Sectors are organized into a number of concentric circles or tracks. As one moves out from the center of the disk, -the tracks get larger. Some disks store the same number of sectors on each track, with outer tracks being recorded using lower bit densities. Other disks place more sectors on outer tracks. On such a disk, more information can be accessed from an outer track than an inner one during a single rotation of the disk.

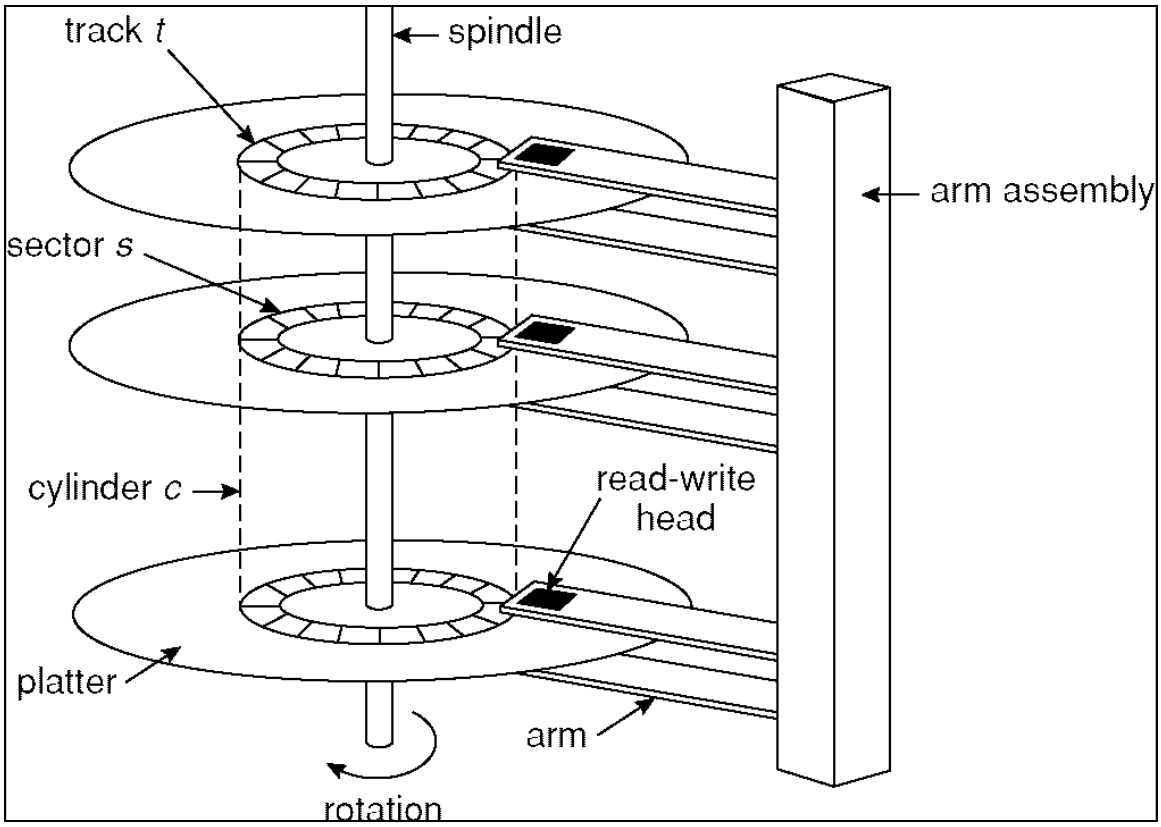


Figure 1: Moving head disk mechanism

The disk speed is composed of three parts:

(a) Seek Time

(b) Latency Time

(c) Transfer time

7.2.1.1 Seek time

To access a block from the disk, first of all the system has to move the read/write head to the required position. The time consumed in this operation is known as seek time and the head movement is called seek. Seek time (S) is determined in terms of:

I: startup delays in initiating head movement.

H: the rate at which read/write head can be moved.

C: How far the head must travel.

S = H * C + I

7.2.1.2 Latency time

Once the head is positioned at the right track, the disk is to be rotated to move the desired block under the read/write head. This delay is known as latency time. On average this will be one-half of one revolution. Thus latency can be computed by dividing the number of revolution per minute (RPM), R, into 30.

$L = 30 / R$

7.2.1.3 Transfer time

Finally the actual data is transferred from the disk to main memory. The time consumed in this operation is known as transfer time. Transfer time T, is determined by the amount of information to be read, B; the number of bytes per track, N; and the rotational speed.

$T = 60B/RN$

So the total time (A) to service a disk request is the sums of these three i.e. seek time, latency time, and transfer time.

$A = S + L + T$

Since most of the systems depend heavily on the disk, so it become very important to make the disk service as fast as possible.

So a number of variations have been observed in disk organization motivated by the desire to reduce the access time, increase the capacity of the disk and to make optimum use of disk surface. For example there may be one head for every track on the disk surface. Such arrangement is known as fixed-head disk. In this arrangement it is very easy for computer to switch from one track to another quickly but it makes the disk very expensive due to the requirement of a number of heads. Generally there is one head that moves in and out to access different tracks because it is cheaper option.

Higher disk capacities are obtained by mounting many platters on the same spindle to form a disk pack. There is one read/write head per circular surface of a platter. All heads of the disk are mounted on a single disk arm, which moves radially to access different tracks. Since the heads are located on the identically positioned tracks of different surfaces, so such tracks can be accessed without any further seeks. So placing that data in one cylinder can speed up sequential

access. Cylinder is a collection of identically positioned tracks of different surfaces.

The hardware for a disk system can be divided into two parts. The disk drive is the mechanical part, including the device motor, the read/write heads and associated logic. The other part called the disk controller determines the logical interaction with the computer. The controller takes instructions from the CPU and orders the disk drive to carry out the instructions.

Every disk drive has a queue of pending requests to be serviced. Whenever a process needs I/O to or from the disk, it issues a request to the operating system, which is placed in the disk queue. The request specifies the disk address, memory address, amount of information to be transferred, and the type of operation (input or output).

7.2.2 Disk Scheduling

For a multiprogramming system with many processes, the disk queue may often be non-empty. Thus, when a request is complete, the disk scheduler has to pick a new request from the queue and service it. As apparent, the amount of head movement needed to satisfy a series of I/O requests could affect the performance. For this reason, a number of scheduling algorithms have been proposed.

7.2.2.1 First cum first served (FCFS) scheduling

This form of scheduling is the simplest one but may not provide the best service. The algorithm is very easy to implement. In it the system picks every time the first request from the disk queue. In this scheduling the total seek time may be substantially high as evident from the following example:

Considered an ordered disk queue with requests involving tracks:

86,140, 23, 50, 12, 89, 14, 120, 64

The following figure shows the movement of read/write head in First Come First Serve scheduling.

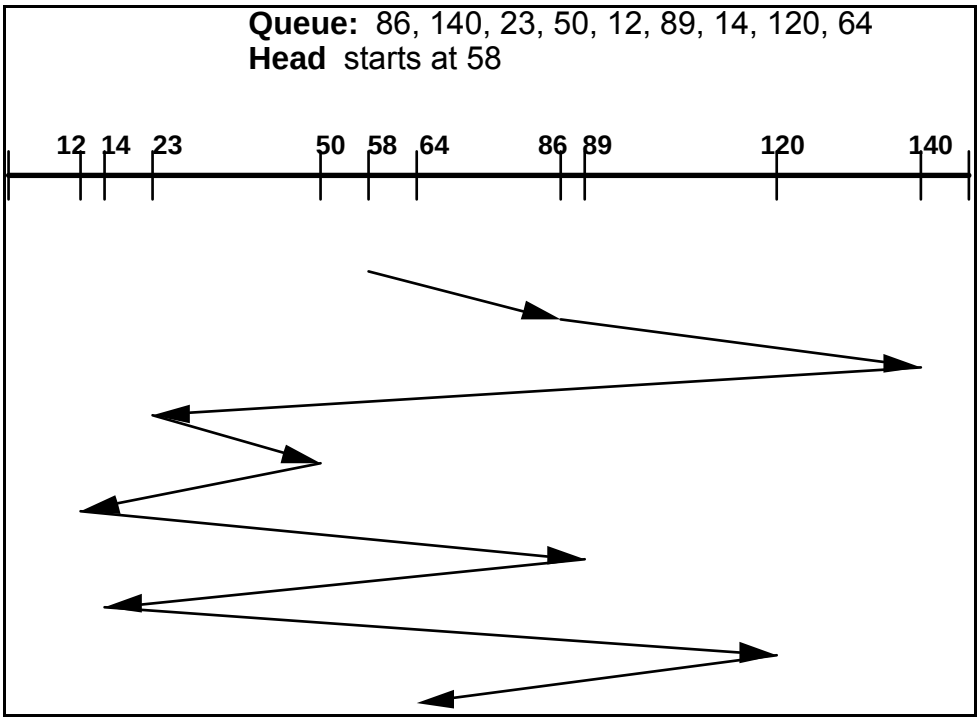


Figure 2: First Come First Serve Disk scheduling

As evident from the above figure, a lot of time (i.e. seek time) is consumed in to and fro movement of the head.

7.2.2.2 Shortest seek time first (SSTF) scheduling

This scheduling algorithm services the request whose track position is closest to the current track position. Shortest-Seek-Time-First selects the request that is asking for minimum seek time from the current head position. Since seek time is generally proportional to the track difference between the requests, this approach is implemented by moving the head to the closest track in the request queue.

The following figure shows the read/write head movement in Shortest-Seek-Time-First scheduling for the above example discussed in First Come First Serve scheduling. It shows a substantial improvement in disk services i.e. reduction in the total movement of the head resulting into the reduced seek time.

Shortest Seek Time First is just like the Shortest Job First process scheduling. So it is having the limitations of Shortest Job First also. It may cause starvation of some requests.

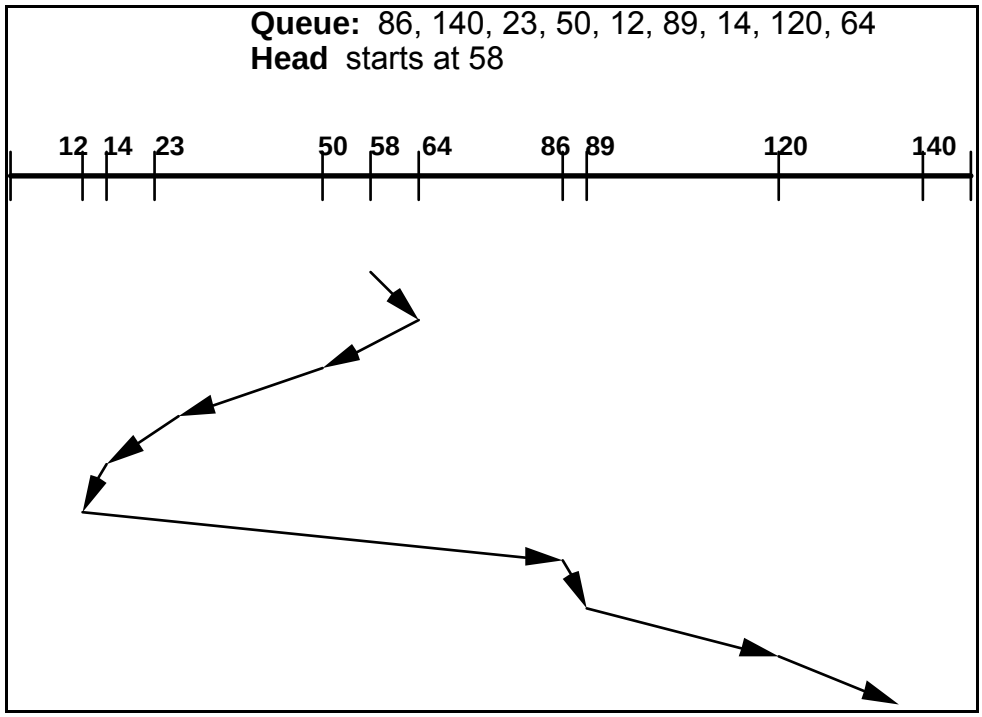


Figure 3: Shortest Seek Time First Scheduling

7.2.2.3 Scan

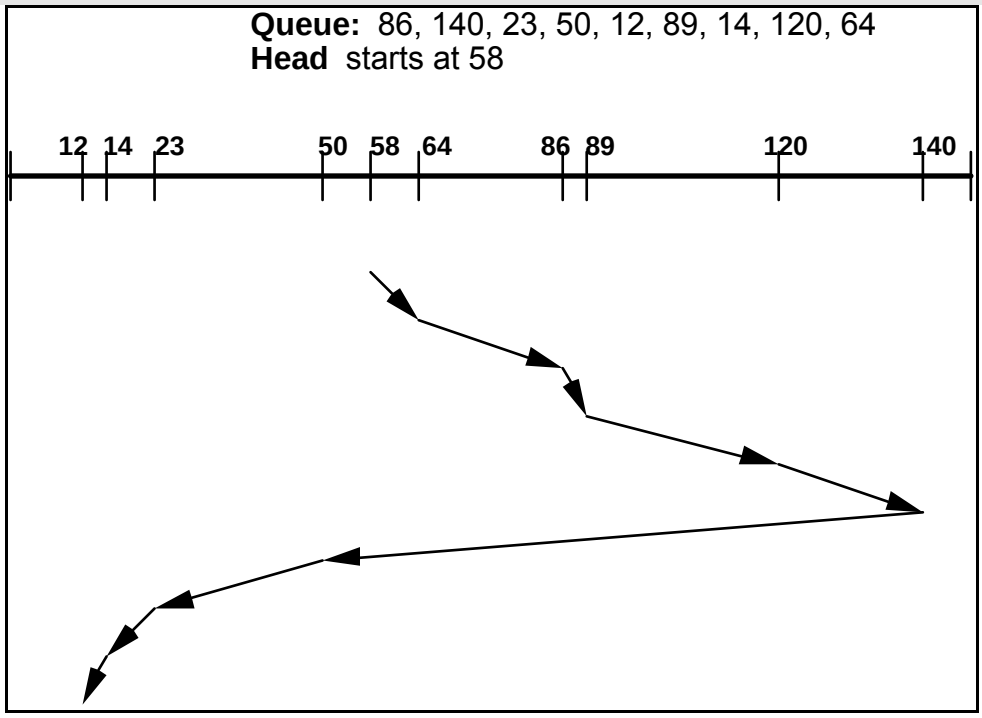


Figure 4: Scan scheduling

In this algorithm the read/write head moves back and forth between the innermost and outermost tracks. As the head gets to each track, satisfies all outstanding requests for that track. In this algorithm also, starvation is possible only if there are repeated requests for the current track. The scan algorithm is sometimes called the elevator algorithm. As it is familiar to the behavior of elevators as they service requests to move from floor to floor in a building.

X.2.2.4 C-scan (Circular scan)

C-scan is a variant of scan. It is designed to provide a more uniform wait time. C-scan moves the head from one end of the disk to another, servicing requests as it goes. When it reaches the other end, however, it immediately return to the beginning of the disk, without servicing any requests on the return trip. C-scan treats the disk, as it was circular, with the last track adjacent to the first one.

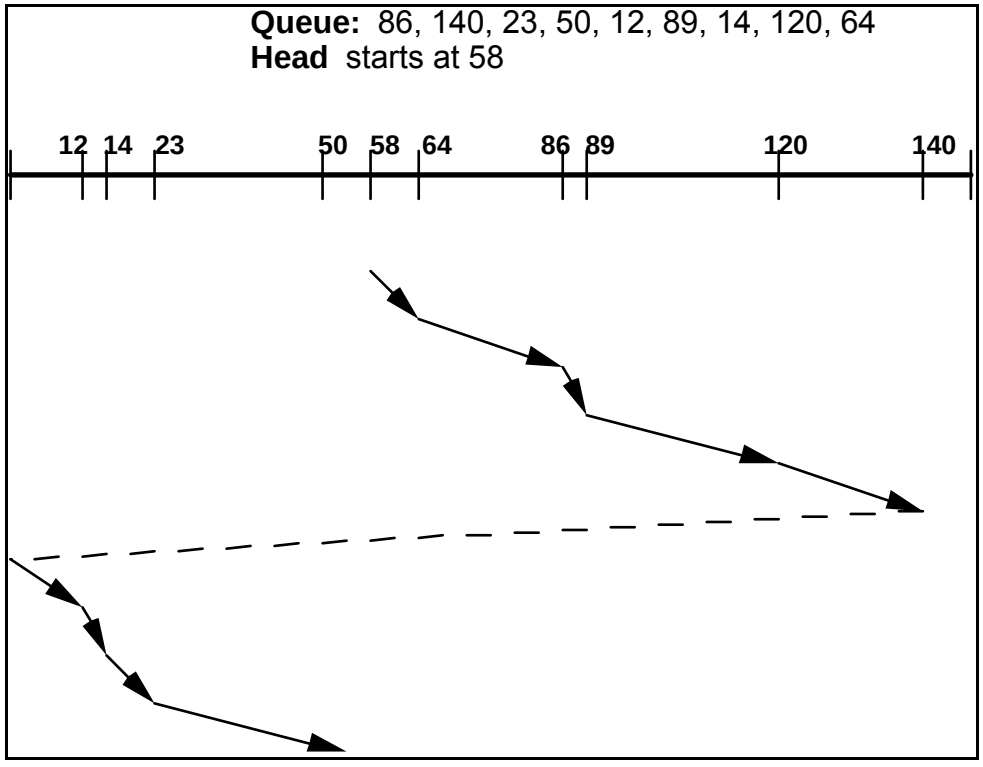


Figure 5: C-Scan scheduling

7.2.2.5 Look

This algorithm is also similar to scan but unlike scan, the head does not unnecessarily travel to the innermost track and outermost track on each circuit. In

this algorithm, head moves in one direction, satisfying the request for the closest track like scan in that direction. When there are no more requests in that direction the head is traveling, head reverse the direction and repeat.

7.2.2.6 N-step scan

In it the request queue is divided into sub queues with each sub queue having a maximum length of N. Sub queues are processed in FIFO order. Within a sub queue, requests are processed using Scan. While a sub queue is being serviced, incoming requests are placed in the next non-filled sub queue. N-step scan eliminates any possibility of starvation.

7.2.2.7 F-Scan

The "F" stands for "freezing" the request queue at a certain time. It is just like N-step scan but there are two sub queues only and each is of unlimited length. While requests in one sub queue are serviced, new requests are placed in other sub queue.

7.2.3 Scheduling algorithm selection

As there are so many disk-scheduling algorithms, an important question is how to choose a scheduling algorithm that will optimize the performance. The commonly used algorithm is Shortest-Seek-Time-First and it has a natural appeal also. San and its variants are more appropriate for system with a heavy load on the disk. It is possible to define an optimal scheduling algorithm, but computational overheads required for that may not justify the savings over Shortest-Seek-Time-First and scan.

No doubt in any scheduling algorithm the performance depends on the number and types of the requests. If every time there is only one outstanding request, then the performance of all the scheduling algorithms will be more or less equivalent. Studies suggest that even First-Come-First-Serve performance will also be reasonably well.

It is also observed that performance of scheduling algorithms is also greatly influenced by the file allocation method. The requests generated by the

contiguously allocated files will result in minimum movement of the head. But in case of indexed access and direct access where the blocks of a file are scattered on the disk surface resulting into a better utilization of the storage space, there may be a lot of movement of the head.

In all these algorithms, to improve the performance, the decision is taken on the basis of head movement i.e. seek time. Latency time is not considered as a factor. Because it is not possible to predict latency time because rotational location cannot be determined. However, multiple requests for the same track may be serviced based on latency.

7.2.4 Sector queuing

In case of disk with one read/write head, the objective of the entire scheduling algorithm is to minimize the seek time by minimizing the movement of read/write head. The dick scheduling algorithms like First Come First Serve, Shortest-Seek-Time-First, Scan, and C-scan center on this objective. But in case of the storage devices such as drum, which has the fixed head, this is not an issue at all. So different scheduling algorithms is used for this type of devices known as sector queuing.

In fixed-head storage devices the tracks are divided into a fixed number of blocks, known as sectors. Any I/O requests specify the address composed of track number and sector number. As seek time is zero for fixed-head storage devices, the important issue is the latency time to improve the performance.

In sector queuing a separate queue is maintained for each sector of the drum. When a request arrives for sector i, it is placed in the queue for sector i. As sector i rotates beneath the read/write head, the first request in its queue is serviced.

Formatting

Before data can be written to a disk, all the administrative data must be written to the disks, organizing it into sectors. This low-level formatting or physical formatting is often done by the manufacturer. In the formatting process, some

sectors may be found to be defective. Most disks have spare sectors, and a remapping mechanism substitutes spare sectors for defective ones.

For sectors that fail after formatting, the operating system may implement a bad block mechanism. Such mechanisms are usually in terms of blocks and are implemented at a level above the device driver. The manner in which sectors are positioned on a track can affect disk performance. If disk I/O operations are limited to transferring a single sector at a time, to read multiple sectors in sequence, separate I/O operations must be performed. After the first I/O operation completes, its interrupt must be processed and the second I/O operation must be issued. During this time, the disk continues to spin. If the start of the next sector to be read has already spun past the read/write head, the sector cannot be read until the next revolution of the disk brings the cylinder by the read/write head. In a worst-case scenario, the disk must wait almost a full revolution. To avoid this problem, sectors may be interleaved. The degree of interleaving is determined by how far the disk revolves in the time from the end of one I/O operation until the controller can issue a subsequent I/O operation. The sector layout, given different degrees of interleaving, is illustrated in Fig. 6.

For most modem hard-disk controllers, interleaving is not used. The controller contains sufficient memory to store an entire track, so a single I/O operation may be used to read all the sectors on a track. Interleaving is more commonly used on less sophisticated disk systems, like floppy disks.

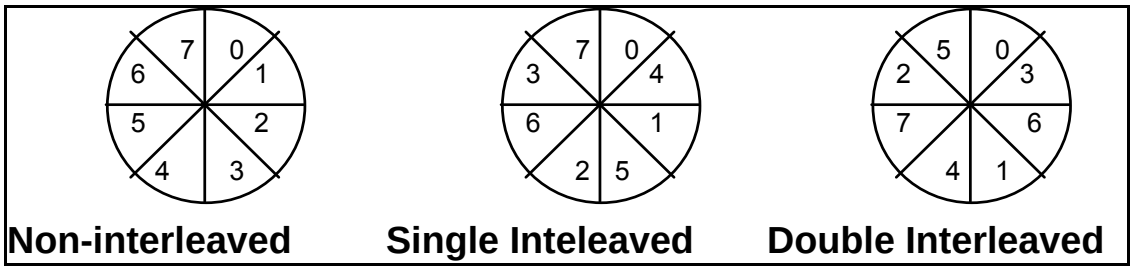


Figure 6: Interleaving

Most operating systems also provide the ability for disks to be divided into one or more virtual disks called partitions. On personal computers, Unix, Windows, and DOS all adhere to a common partitioning scheme so that they all may co-reside on a single disk.

Before files can be written to a disk, an empty file system must be created on a disk partition. This requires the various data structures required by the file system to be written to the partition, and is known as a high-level format or logical format. A file system is not required to make use of a disk. Some operating systems allow applications to write directly to a disk device. To an application, a directly accessible disk device is just a large sequential collection of storage blocks. In such a case, it is the responsibility of the application to impose an order on the information written there.

7.2.5 RAID (Redundant Array of Inexpensive Disks)

RAID is the organization of multiple disks into a large, high performance logical disk. Disk arrays stripe data across multiple disks and access them in parallel to achieve higher data transfer rates on large data accesses and higher I/O rates on small data accesses.

Data striping also results in uniform load balancing across all of the disks, eliminating hot spots that otherwise saturate a small number of disks, while the majority of disks sit idle. But large disk arrays, however are highly vulnerable to disk failures. So the solution to the problem of lower reliability (Reliability is how well a system can work without any failures in its components) in disk arrays is to improve the availability (Availability is how well a system can work in times of a failure. If a system is able to work even in the presence of a failure of one or more system components, the system is said to be available) of the system. This

can be achieved by employing redundancy in the form of error-correcting codes to tolerate disk failures. But redundancy has its own limitations. Every time there is a write operation, this change has to be reflected in the disks storing redundant information, making write operation a time consuming one. Also, keeping the redundant information consistent in the presence of concurrent I/O operation can be difficult.

The Need for RAID

The two keywords, Redundant and Array, are self explanatory.

- An array of multiple disks accessed in parallel will give greater throughput.
- Redundant data on multiple disks provides fault tolerance.

With a single hard disk, you cannot protect yourself against the costs of a disk failure, the time required to obtain and install a replacement disk, reinstall the operating system, restore files from backup tapes, and repeat all the data entry performed since the last backup was made. With multiple disks and a suitable redundancy scheme, your system can stay up and running when a disk fails, and even while the replacement disk is being installed and its data restored.

In RAID configuration, we need to simultaneously achieve the following goals:

- Maximize the number of disks being accessed in parallel.
- Minimize the amount of disk space being used for redundant data.
- Minimize the overhead required to achieve the above goals.

Data striping and Redundancy

There are 2 important concepts to be understood in the design and implementation of disk arrays: (1) Data striping, for improved performance & (2) Redundancy for improved availability. Data striping transparently distributes data over multiple disks to make them appear as a single fast, large disk and improves aggregate I/O performance by allowing multiple I/Os to be serviced in parallel. There are 2 aspects to this parallelism.

- Multiple, independent requests can be serviced in parallel by separate disks. This decreases the queuing time seen by I/O requests.
- Single, multiple block requests can be serviced by multiple disks acting in co-ordination. This increases the effective transfer rate seen by a single request.

Most of the redundant disk array organizations can be distinguished based on 2 features:

1. The granularity of data interleaving and
2. The way redundant data is computed & stored across the disk array.

Data interleaving can be either fine grained or coarse grained. Fine grained disk arrays conceptually interleave data in relatively small units so that all I/O requests, regardless of their size, access all of the disks in the disk array. This results in very high data transfer rate for all I/O requests but has the disadvantages that only one logical I/O request can be in service at any given time and all disks must waste time positioning for every request. Coarse grained disk arrays interleave data in relatively large units so that small I/O requests need access only a small number of disks while large requests can access all the disks in the disk array. This allows multiple small requests to be serviced simultaneously while still allowing large requests to see the higher transfer rates afforded by using multiple disks.

Redundancy

To improve the reliability of the array of disks, redundancy is incorporated in the array of disks. This redundancy brings up two problems:

1. Selecting the method for computing the redundant information.
2. Selecting a method for distribution of the redundant information across the disk array.

The distribution method can be classified into 2 different schemes:

- Schemes that concentrate redundant information on a small number of disks.
- Schemes that distribute redundant information uniformly across all of the disks.

It is worth to mention that selecting between the many possible data striping and redundancy schemes involves complex tradeoffs between availability, performance and cost. There are many types of RAID organizations which are given below:

7.2.5.1 Non-Redundant Disk array (RAID Level 0)

RAID level 0 has the minimum cost because it does not employ redundancy at all so it never needs to update redundant information. But it does not have the best performance. Redundancy schemes can perform better on reads by selectively scheduling requests on the disk with the shortest expected seek and rotational delays. Without, redundancy, any single disk failure will result in data-loss

The size of a data block i.e. "stripe width", varies with the implementation, but is always at least as large as a disk's sector size. Sequential blocks of data are written across multiple disks in stripes, as follows:

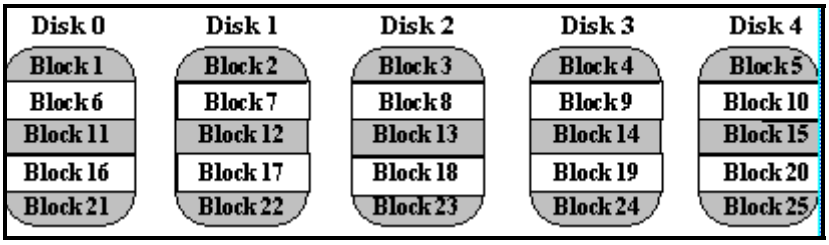


Figure 7 - RAID 0

7.2.5.2 Mirrored (RAID Level 1)

The mirroring uses twice as many disks as RAID Level 0. Whenever data is written to a disk the same data is also written to a redundant disk, so that there are always two copies of the information. When data is read, it can be retrieved from the disk with the shorter queuing, seek and rotational delays. If a disk fails, the other copy is used to service requests.

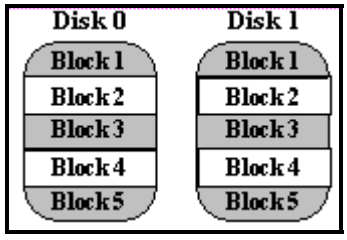


Figure 8 – RAID 1

7.2.5.3 Memory-Style (RAID Level 2)

Memory systems have provided recovery from failed components with much less cost than mirroring by using Hamming codes. Hamming codes contain parity for distinct overlapping subsets of components. In one version of this scheme, four disks require three redundant disks, one less than mirroring. Since the number of redundant disks is proportional to the log of the total number of the disks on the system, storage efficiency increases as the number of data disks increases.

If a single component fails, several of the parity components will have inconsistent values, and the failed component is the one held in common by each incorrect subset. The lost information is recovered by reading the other components in a subset, including the parity component, and setting the missing bit to 0 or 1 to create proper parity value for that subset. Thus, multiple redundant disks are needed to identify the failed disk, but only one is needed to recover the lost information.

A RAID 2 system would normally have as many data disks as the word size of the computer, typically 32. In addition, RAID 2 requires the use of extra disks to store an error-correcting code for redundancy. With 32 data disks, a RAID 2 system would require 7 additional disks for a Hamming-code ECC.

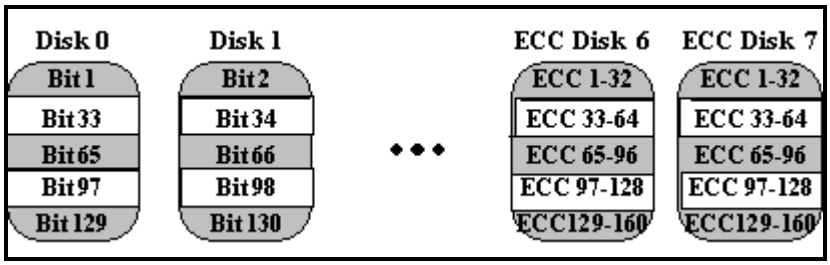


Figure 9 – RAID 2

7.2.5.4 Bit-Interleaved Parity (RAID Level 3)

In this scheme the parity disk is written in the same way as the parity bit in normal Random Access Memory (RAM), where it is the Exclusive Or of the 8, 16 or 32 data bits. In RAM, parity is used to detect single-bit data errors, but it cannot correct them because there is no information available to determine which bit is incorrect. With disk drives, however, we rely on the disk controller to report a data read error. Knowing which disk's data is missing, we can reconstruct it as the Exclusive Or (XOR) of all remaining data disks plus the parity disk.

In a bit-interleaved, parity disk array, data is conceptually interleaved bit-wise over the data disks, and a single parity disk is added to tolerate any single disk failure. Each read request accesses all data disks and each write request accesses all data disks and the parity disk. Thus, only one request can be serviced at a time. Because the parity disk contains only parity and no data, the parity disk cannot participate on reads, resulting in slightly lower read performance than for redundancy schemes that distribute the parity and data

over all disks. Bit-interleaved, parity disk arrays are frequently used in applications that require high bandwidth but not high I/O rates. They are also simpler to implement than RAID levels 4, 5, and 6.

Here, the parity disk is written in the same way as the parity bit in normal Random Access Memory (RAM), where it is the Exclusive Or of the 8, 16 or 32 data bits. In RAM, parity is used to detect single-bit data errors, but it cannot correct them because there is no information available to determine which bit is incorrect. With disk drives, however, we rely on the disk controller to report a data read error. Knowing which disk's data is missing, we can reconstruct it as the Exclusive Or (XOR) of all remaining data disks plus the parity disk.

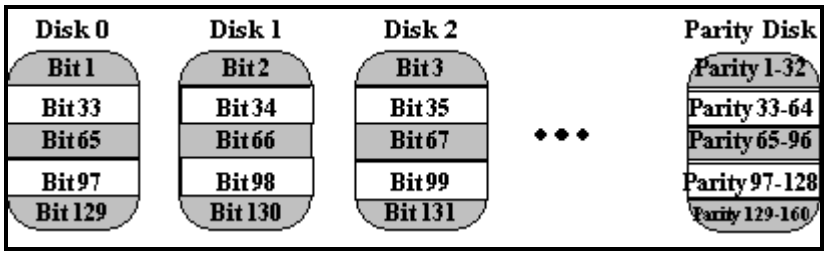


Figure 10 – RAID 3

7.2.5.5 Block-Interleaved Parity (RAID Level 4)

The block-interleaved, parity disk array is similar to the bit-interleaved, parity disk array except that data is interleaved across disks of arbitrary size rather than in bits. The size of these blocks is called the striping unit. Read requests smaller than the striping unit access only a single data disk. Write requests must update the requested data blocks and must also compute and update the parity block. For large writes that touch blocks on all disks, parity is easily computed by exclusive-oring the new data for each disk. For small write requests that update only one data disk, parity is computed by noting how the new data differs from the old data and applying those differences to the parity block. Small write requests thus require four disk I/Os: one to write the new data, two to read the old data and old parity for computing the new parity, and one to write the new parity. This is referred to as a read-modify-write procedure. Because a block-interleaved, parity disk array has only one parity disk, which must be updated on all write operations, the parity disk can easily become a bottleneck. Because of

this limitation, the block-interleaved distributed parity disk array is universally preferred over the block-interleaved, parity disk array.

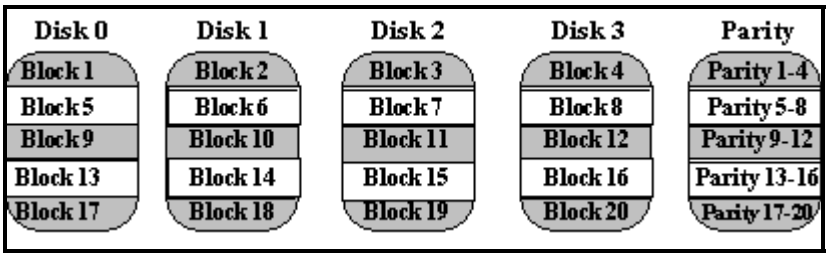


Figure 11 – RAID 4

7.2.5.6 Block-Interleaved Distributed-Parity (RAID Level 5)

The block-interleaved distributed-parity disk array eliminates the parity disk bottleneck present in the block-interleaved parity disk array by distributing the parity uniformly over all of the disks. An additional, frequently overlooked advantage to distributing the parity is that it also distributes data over all of the disks rather than over all but one. This allows all disks to participate in servicing read operations in contrast to redundancy schemes with dedicated parity disks in which the parity disk cannot participate in servicing read requests. Block-interleaved distributed-parity disk array have the best small read, large write performance of any redundancy disk array. Small write requests are somewhat inefficient compared with redundancy schemes such as mirroring however, due to the need to perform read-modify-write operations to update parity. This is the major performance weakness of RAID level 5 disk arrays.

The exact method used to distribute parity in block-interleaved distributed-parity disk arrays can affect performance. Following figure illustrates left-symmetric parity distribution.

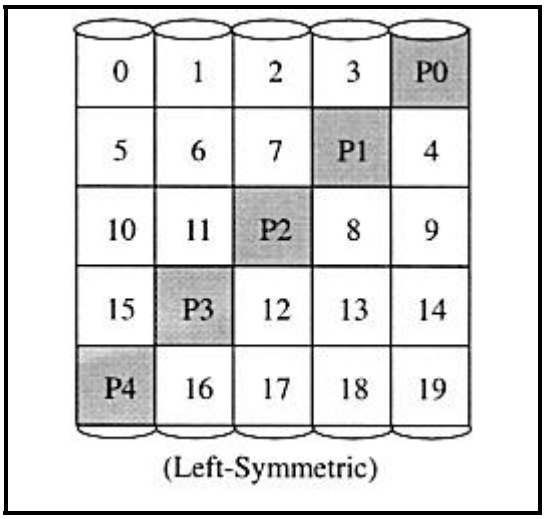


Figure 12

Each square corresponds to a stripe unit. Each column of squares corresponds to a disk. P0 computes the parity over stripe units 0, 1, 2 and 3; P1 computes parity over stripe units 4, 5, 6, and 7 etc. A useful property of the left-symmetric parity distribution is that whenever you traverse the striping units sequentially, you will access each disk once before accessing any disk device. This property reduces disk conflicts when servicing large requests.

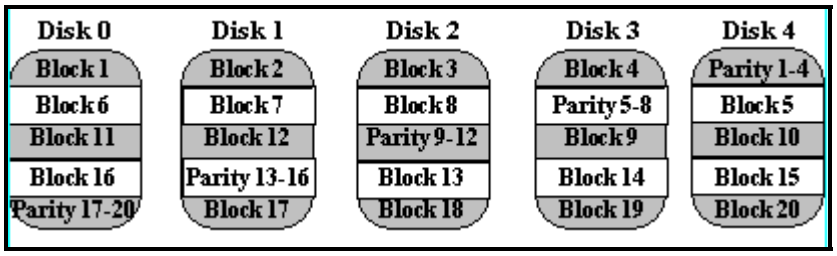


Figure 13 – RAID 5

7.2.5.7 P+Q redundancy (RAID Level 6)

Parity is a redundancy code capable of correcting any single, self-identifying failure. As large disk arrays are considered, multiple failures are possible and stronger codes are needed. Moreover, when a disk fails in parity-protected disk array, recovering the contents of the failed disk requires successfully reading the contents of all non-failed disks. The probability of encountering an uncorrectable read error during recovery can be significant. Thus, applications with more stringent reliability requirements require stronger error correcting codes.

Once such scheme, called P+Q redundancy, uses Reed-Solomon codes to protect against up to two disk failures using the bare minimum of two redundant disk arrays. The P+Q redundant disk arrays are structurally very similar to the

block-interleaved distributed-parity disk arrays and operate in much the same manner. In particular, P+Q redundant disk arrays also perform small write operations using a read-modify-write procedure, except that instead of four disk accesses per write requests, P+Q redundant disk arrays require six disk accesses due to the need to update both the `P' and `Q' information.

7.2.5.8 Striped Mirrors (RAID Level 10)

RAID 10 was not mentioned in the original 1988 article that defined RAID 1 through RAID 5. The term is now used to mean the combination of RAID 0 (striping) and RAID 1 (mirroring). Disks are mirrored in pairs for redundancy and improved performance, then data is striped across multiple disks for maximum performance. In the diagram below, Disks 0 & 2 and Disks 1 & 3 are mirrored pairs.

Obviously, RAID 10 uses more disk space to provide redundant data than RAID 5. However, it also provides a performance advantage by reading from all disks in parallel while eliminating the write penalty of RAID 5. In addition, RAID 10 gives better performance than RAID 5 while a failed drive remains unreplaced. Under RAID 5, each attempted read of the failed drive can be performed only by reading all of the other disks. On RAID 10, a failed disk can be recovered by a single read of its mirrored pair.

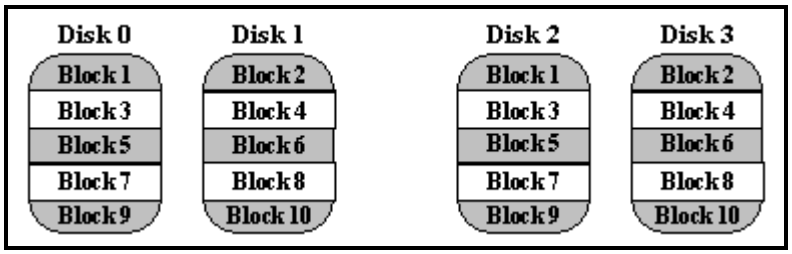


Figure 14

7.3 Summary

As processor and main memory speeds increase more rapidly than those of secondary storage devices, *optimizing disk performance has become important* to realize optimal performance. As the platters of disk spin, each read-write head sketches out a circular track of data on a disk surface to access. The time it takes for the head to move from its current cylinder to the one containing the data record being accessed is called the seek time. The time it takes for data to rotate

from its current position to a position adjacent to the read/write head is called latency time. Many processes can generate requests for reading and writing data on a disk simultaneously. Because these processes sometimes make requests faster than they can be serviced by the disk, queues build up to hold disk requests. Some early computing systems simply serviced these requests on a first-come-first-served (FCFS) basis which is a fair method, but when the request rate becomes heavy, FCFS results in long waiting times. To reduce the time spent seeking records, it seems reasonable to order the request queue in some other manner. This reordering is called disk scheduling. The two most common types of scheduling are seek optimization and rotational optimization which are evaluated by comparing their throughput, mean response time and variance of response times.

Shortest-seek-time-first (SSTF) scheduling services the request that is closest to the read-write head's current cylinder. By reducing average seek times, SSTF achieves higher throughput rates than FCFS, and mean response times tend to be lower for moderate loads. The SCAN scheduling strategy reduces unfairness and variance of response times by choosing the request that requires the shortest seek distance in a preferred direction. Thus, if the preferred direction is currently outward, the SCAN strategy chooses the shortest seek distance in the outward direction. However, because SCAN ensures that all requests in a given direction will be serviced before the requests in the opposite direction, it offers a lower variance of response times than SSTF.

The circular SCAN (C-SCAN) modification to the SCAN strategy moves the arm from the outer cylinder to the inner cylinder, servicing requests on a shortest-seek basis. When the arm has completed its inward sweep, it jumps to the outermost cylinder, then resumes its inward sweep processing requests. C-SCAN maintains high levels of throughput while further limiting variance of response times by avoiding the discrimination against the innermost and outermost cylinders.

The FSCAN and N-Step SCAN modifications to the SCAN strategy eliminate the possibility of indefinitely postponing requests. FSCAN uses the SCAN strategy to

service only those requests waiting when a particular sweep begins. Requests arriving during a sweep are grouped together and ordered for optimum service during the return sweep. N-Step SCAN services the first n requests in the queue using the SCAN strategy. When the sweep is complete, the next n requests are serviced. Arriving requests are placed at the end of the request queue, which prevents requests in the current sweep from being indefinitely postponed. The LOOK variation of the SCAN strategy "looks" ahead to the end of the current sweep to determine the next request to service. If there are no more requests in the current direction, LOOK changes the preferred direction and begins the next sweep, stopping when passing a cylinder that corresponds to a request in the queue. This strategy eliminates unnecessary seek operations experienced by other variations of the SCAN strategy by preventing the read/write head from moving to the innermost or outermost cylinders unless it is servicing a request to those locations.

The circular LOOK (C-LOOK) variation of the LOOK strategy uses the same technique as C-SCAN to reduce the bias against requests located at extreme ends of the platters. When there are no more requests on a current sweep, the read/write head moves to the request closest to the outer cylinder and begins the next sweep. Sector queuing is a scheduling algorithm for fixed head devices such as drums.

RAID is the organization of multiple disks to achieve higher data transfer rates on large data accesses and higher I/O rates on small data accesses. Data striping also results in uniform load balancing across all of the disks. To cope with the problem of reliability, redundancy is introduced which has its own limitations at the time of write operations. A number of RAID organizations are available with their merits and demerits.

7.4 Keywords

- **Seek time**: To access a block from the disk, first of all the system has to move the read/write head to the required position. The time consumed in moving the head to access a block from the disk is known as seek time.

- **Latency time:** the time consumed in rotating the disk to move the desired block under the read/write head.
- **Transfer time:** The time consumed in transferring the data from the disk to the main memory is known as transfer time.
- **RAID:** It is the Redundant Array of Inexpensive Disks, an organization of multiple disks into a large, high performance logical disk.

7.5 SELF ASSESMENT QUESTIONS (SAQ)

1. Compare the throughput of scan and C-scan assuming a uniform distribution of requests.
2. What do you understand by seek time, latency time, and transfer time? Explain.
3. Shortest Seek Time First favors tracks in the center of the disk. On an operating system using Shortest Seek Time First, how might this affect the design of the file system?
4. When there is no outstanding request in the queue, all the disk-scheduling algorithms reduce to First Come First Serve scheduling? Explain why?
5. The entire disk scheduling algorithms except First Come First Serve may cause starvation and hence not truly fair.
    - Explain why.
    - Come up with a scheme to ensure fairness.
    - Why is fairness an important goal in a time-sharing system?
6. What do you understand by RAID? What are the objectives of it? Explain.
7. What are the limitations due to redundancy in RAID? What are the advantages of redundancy? Explain.
8. Write a detailed note on different RAID organizations? Discuss their merits and demerits also.

7.6 SUGGESTED READINGS / REFERENCE MATERIAL

1. Operating System Concepts, 5th Edition, Silberschatz A., Galvin P.B., John Wiley & Sons.
2. Systems Programming & Operating Systems, 2nd Revised Edition, Dhamdhere D.M., Tata McGraw Hill Publishing Company Ltd., New Delhi.

3. Operating Systems, Madnick S.E., Donovan J.T., Tata McGraw Hill Publishing Company Ltd., New Delhi.
4. Operating Systems-A Modern Perspective, Gary Nutt, Pearson Education Asia, 2000.
5. Operating Systems, Harris J.A., Tata McGraw Hill Publishing Company Ltd., New Delhi, 2002.

Lesson number: 8	***Writer: Dr. Rakesh Kumar***
Network & Distributed Operating System	***Vetter: Dr. Pradeep Bhatia***

8.0 Objectives

The objectives of this lesson are to identify the factors motivating the use and development of distributed and network operating systems and make the students familiar with the important issues involved in the development of network and distributed operating systems.

8.1 Introduction

The inherent evolution of more powerful processors does not to satisfy the demand for computation. Most of those computations are highly dependant on time. We do not want to wait for an email to arrive at the same time as normal mail would or a certain simulation to finish till a competitor has already done it and started to ship out the resulting product. As we can see from those examples, the computational power needs to meet commercial and time constraints. Not the overall average in computational power is important, but how deadlines require peak times and how the system scales to such events. Although there are examples where response time does not play the crucial role, those applications are seldom. In applications where a missing of a deadline directly or indirectly leads to a human catastrophe, time is a hard constraint. Those hard real-time constraints are getting more important as computer assist more and more our daily life. Two approaches are taken to meet the demand for computations. One is the steady improvement of hardware. CPU improves in an incredible speed, but to do so is quite costly and meets physical barriers. The second one tries to relax this situation by working with more than one CPU on a problem in parallel. It is useful to distinguish two kinds of CPU interconnections in which the response time and data transfer rate are the differing factors. Loosely coupled CPUs are those, which work together and possible communication is slow, distant and more unreliable. This setup called multicomputer and make up distributed systems. An example would be computers connected via modem or

LAN. Multiprocessors on the other hand are closely coupled CPUs with a high bandwidth and response time often working on a shared memory. The communication is based on a multi processor board. These systems are not considered to be distributed systems, although it is not uncommon that they are used in a distributed environment. Some systems are inherently distributed. An example is the inventory and store management of a company. Stores are located all over the country and loosely interconnected with each other to be able to perform global optimization. It is hardly feasible to centralize this structure.

8.2 Presentation of Contents

8.2.1 Evolutions of Modern operating systems

8.2.2 Networking

8.2.2.1 Network Topologies

8.2.3 Types of Networks

8.2.3.1 LAN - Local Area Network

8.2.3.2 WAN - Wide Area Network

8.2.4 Network Operating System

8.2.5 Distributed Operating System

8.2.5.1 Motivations for distributed systems

8.2.5.2 Distributed system architectures

8.2.5.3 Distributed Computation Paradigm

8.2.5.3.1 Data and Computation Migration

8.2.5.3.2 Client-server Computing

8.2.5.3.3 Remote Procedure Call (RPC)

8.2.5.3.4 Remote evaluation

8.2.5.4 Important aspects of distributed systems

8.2.5.4.1 Transparency

8.2.5.4.2 Flexibility

8.2.5.4.3 Reliability

8.2.5.4.4 Performance

8.2.5.4.5 Scalability

8.2.5.4.6 Consistency

8.2.5.4.7 Robustness

8.2.1 Evolutions of Modern operating systems

The following table provides a brief overview of the evolution of modern operating systems.

Generation	System	Characteristics	Goals
First	Centralized Operating System	Process management Memory Management I/O management File management	Resource management, extended machine (Virtuality)
Second	Network operating system	Remote access, information exchange, network browsing	Resource sharing (Interoperability)
Third	Distributed Operating system	Global view of: file system, name space, time, security, computational power	Single computer view of multiple computer systems (Transparency)
Fourth	Cooperative autonomous System	Open and cooperative distributed applications	Cooperative work (Autonomicity)

Network operating systems are straightforward extension of a traditional operating system to facilitate resource sharing and information exchange. Network operating systems are characterized by information exchange divided and implemented at various levels i.e. from communication sub-network to transport services and by inclusion of a transport layer and the support for network applications like remote login, file transfer, messaging, network browsing, and remote execution.

While in Distributed operating systems, sharing of resources and coordination of distributed activities in networked environments are the main goals in the design. The key distinction between a network Operating System and a distributed Operating System is the concept of transparency: concurrency transparency, location transparency, parallelism and performance transparency, migration

transparency, and replication transparency etc. Distributed operating systems consist of three major components:

- Coordination of distributed processes
- Management of distributed resources
- Implementation of distributed algorithms

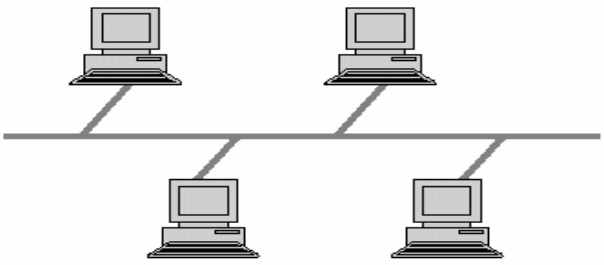
8.2.2 Networking

The key component in network operating system and distributed operating system is network. Computer networking enables its users to share the resources, speed up computations and communicate with other users in the network. The important issues in networking are the types of network, network topologies, connection and routing strategies, and communications protocols.

8.2.2.1 Network Topologies

In computer networking, topology refers to the layout of connected devices. Network topologies are categorized into the following basic types: (a) bus, (b) ring, (c) star, (d) tree, and (e) mesh. More complex networks can be built as hybrids of two or more of the above basic topologies.

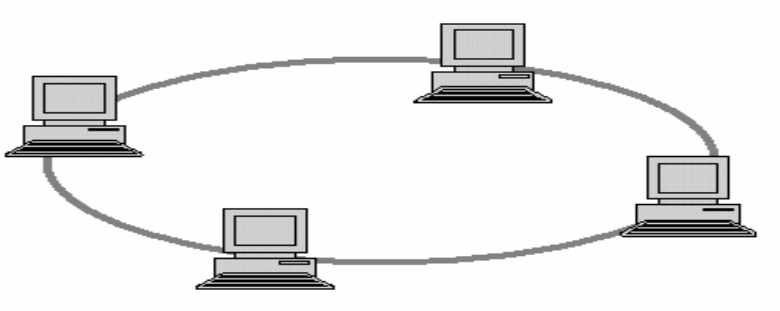
(a) Bus Topology: Bus networks use a common backbone to connect all devices. A single cable i.e. backbone functions as a shared communication medium to which devices are attached with an interface connector. A device wanting to communicate with another device on the network sends a broadcast message onto the wire that all other devices see, but only the intended recipient actually accepts and processes the message.



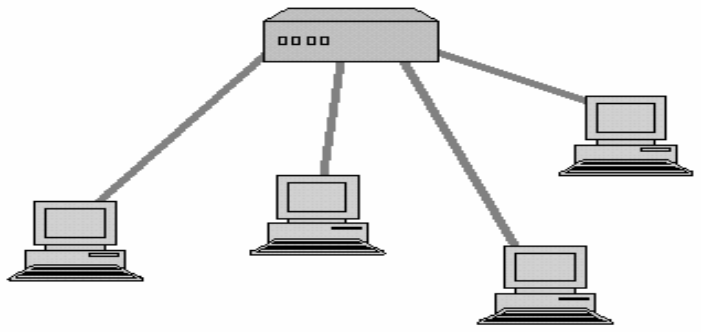
Ethernet bus topologies are relatively easy to install and don't require much cabling compared to the alternatives. 10Base-2 ("ThinNet") and 10Base-5 ("ThickNet") both were popular Ethernet cabling options many years ago for bus topologies. However, bus networks work best with a limited number of devices. If

more than a few dozen computers are added to a network bus, performance problems will likely result. In addition, if the backbone cable fails, the entire network effectively becomes unusable.

(b) Ring Topology: In a ring network, every device has exactly two neighbors for communication purposes. All messages travel through a ring in the same direction. A failure in any cable or device breaks the loop and can take down the entire network. To implement a ring network, one typically uses FDDI, SONET, or Token Ring technology. Ring topologies are found in some office buildings or school campuses.

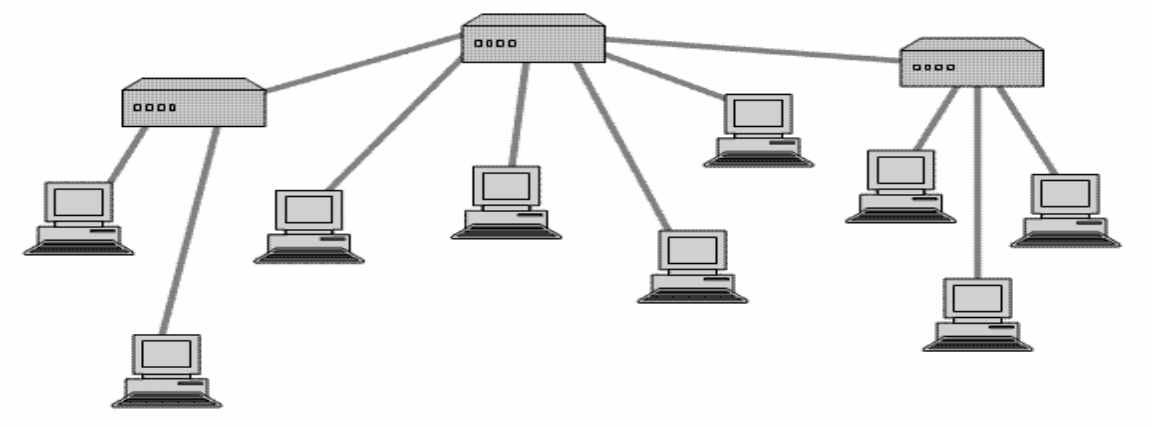


(c) Star Topology: Many home networks use the star topology. A star network features a central connection point called a "hub" that may be a hub, switch or router. Devices typically connect to the hub with Unshielded Twisted Pair (UTP) Ethernet. Compared to the bus topology, a star network generally requires more cable, but a failure in any star network cable will only take down one computer's network access and not the entire LAN.

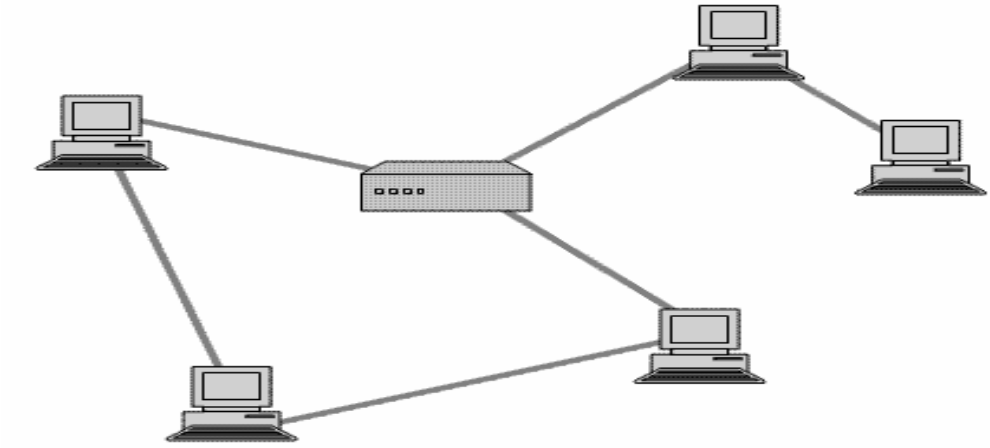


(d) Tree Topology: Tree topologies integrate multiple star topologies together onto a bus. In its simplest form, only hub devices connect directly to the tree bus and each hub functions as the "root" of a tree of devices. This bus/star

hybrid approach supports future expandability of the network much better than a bus or a star alone.



(e) Mesh Topology: Mesh topologies involve the concept of routes. Unlike each of the previous topologies, messages sent on a mesh network can take any of several possible paths from source to destination. Some WANs, most notably the Internet, employ mesh routing. A mesh network in which every device connects to every other is called a full mesh. As shown in the illustration below, partial mesh networks also exist in which some devices connect only indirectly to others.



8.2.3 Types of Networks

Networks have been broadly categorized as LAN and WAN.

8.2.3.1 LAN - Local Area Network

A LAN connects network devices over a relatively short distance. A networked office building, school, or home usually contains a single LAN, though sometimes

one building will contain a few small LANs (perhaps one per room), and occasionally a LAN will span a group of nearby buildings. In TCP/IP networking, a LAN is often but not always implemented as a single IP subnet. In addition to operating in a limited space, LANs are also typically owned, controlled, and managed by a single person or organization. They also tend to use certain connectivity technologies, primarily Ethernet and Token Ring.

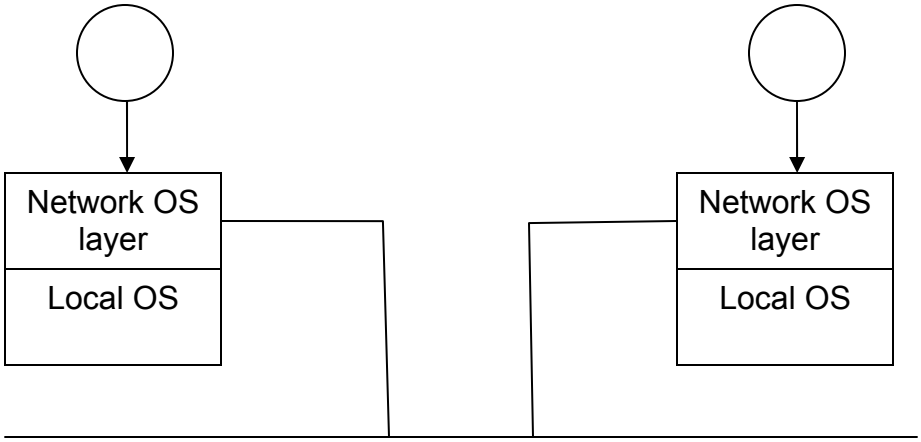
8.2.3.2 **WAN - Wide Area Network**

As the term implies, a WAN spans a large physical distance. The Internet is the largest WAN, spanning the Earth. A WAN is a geographically-dispersed collection of LANs. A network device called a router connects LANs to a WAN. In IP networking, the router maintains both a LAN address and a WAN address. A WAN differs from a LAN in several important ways. Most WANs (like the Internet) are not owned by any one organization but rather exist under collective or distributed ownership and management. WANs tend to use technology like ATM, Frame Relay and X.25 for connectivity over the longer distances.

8.2.4 Network Operating System

The objective of network operating system is to share resources across two or more computer systems each having its own operating system. There is a network Operating System layer between the kernel and the user in network operating system. The processes interact with the network Operating System layer which analyze whether its requirements can be furnished by the local OS, and if not, then it interacts with network Operating System layer of the other node which implements the access to the resource with the help of local Operating System of that node as shown in the following figure.

The main advantage of network operating system is that it is easy to implement it on top of a conventional operating system. But there are demerits also. The two most important requirements of an Operating System are: (a) Facilitate the utilization of resources, (b) Optimize utilization of resources. The Network Operating System satisfies the first requirement but it does not satisfy the second requirement.



Network Operating System

8.2.5 Distributed Operating System

As discussed earlier, the key distinction between a network Operating System and a distributed Operating System is the concept of transparency: concurrency transparency, location transparency, parallelism and performance transparency, migration transparency, and replication transparency

8.2.5.1 Motivations for distributed systems

Following are the factors motivation the use and development of distributed operating systems:

(a) Resource Sharing: Resource sharing is required when application running on one system may require the resources possessed by some other system; such one machine may require the line printer which is attached with some other system.

(b) Computation speed up: The price/service ratio between mainframes and PC workstations has changed. Having 10 machines with 100 MIPS than one with 1000 MIPS has become more economical. But combining these 10 computers puts a burden on the system design. Certain parallel algorithms and computational problems require high constant interaction of CPUs and other do not. So computation speed-up is achieved by decomposing a computation into many sub-computations and spreading these sub-computations across various nodes of the system.

(c) Incremental growth: With distributed architecture it is easy to enhance the capability of the systems. Distributed architecture facilitates the up gradation of existing components as well as addition of new sub-systems.

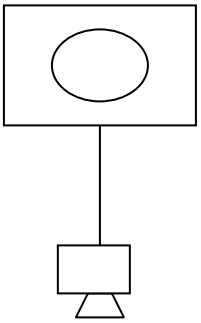
(d) Reliability: Another motivation for distributed systems is reliability. The probability of the system to fail is decreased by involving more than one computer. This redundancy requires the transparency that the different nodes in a cluster are able to take over the work of a failing machine and only jeopardize the quality of service. Developing software is the most complex issue in distributed systems. The correctness of concurrent working processes, the detection of faulty nodes and timing constraints in the network impose serious problems for the system designer. Distributed operating systems should be able to assist the design process by keeping complex tasks of synchronization, optimal process allocation and redundancy away from the programmers. So transparency to the programmer is another requirement for feasible distributed systems. Hiding complexity by transparency becomes necessary in such complex systems.

Although these issues are known for a long time, modern operating systems have huge problems accommodating changes towards a distributed environment. Traditionally single CPU computers did not require complex synchronization and changing these systems to cope with real parallelism has shown to be greatly complicated since starting the design of new operating systems from scratch is hardly feasible because legacy software is still economical to use. This is another reason for transparency. New designed operating systems tend to tackle the problems with small interconnected modules. These are microkernel systems in which each module tries to solve a small part of the big problems. Since development of distributed systems is still an unknown issue, micro-kernels are way more flexible to accommodate changes in the design. This paradigm of modules is expected to creep into monolithic kernels.

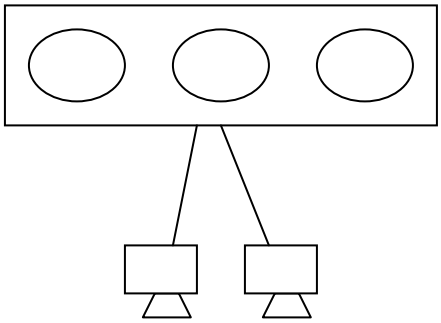
8.2.5.2 Distributed system architectures

Two main different models can be distinguished for sharing CPUs. The workstation and the processor pool model:

Workstation-server model: In it a workstation may serve as a stand-alone computer or as a part of an overall network. In the workstation model the cooperating CPUs are spread over the entire network. Diskless workstations can be used to share common data on a file server while processing is done locally. Workstations with disks can be used to increase reliability of persistent data with a transparent distributed file system. This has proven to be hard and no standard distributed file system exists today which solve all problems of such an approach. Consistency in such a configuration is hard to maintain. Some distributed databases exist which handle those problems with transactions. Atomic transactions are also used to synchronize processes.



Workstation model



Processor-pool model

Processor-pool model: It allows collecting of all processing power in one place and leaving the users with only a terminal. The multiprocessor model of a processor pool is a centralized approach. X window terminals share a pool of processors at a centralized point. The benefit is scalability. As soon as new computational needs arise, additional processors can easily be plugged into the processor board which is shared evenly among all users. Administration of a consistent environment is easily possible. The main disadvantages of this approach are the introduction of a single point of failure and high cost of multiprocessors. For those reasons processor pools are not considered to be distributed systems although it might be argued that the X terminals provide computational services to remote locations. Scheduling decisions are far easier in processor pools. The locations of processes in a SMP (Symmetric Multi-

Processing) system do not play a crucial role. It is only slightly beneficial to place processes on CPU where they ran on before, to avoid unnecessary cache invalidations on the die. In some multiprocessors, NUMA (non-uniform memory allocation) architecture makes it necessary to place processes on processors close to their memory. But this is far less complex that the considerations in the workstation model where heterogeneous architectures have to work together and communication is likely to brake down.

A truly distributed operating system gives the user the illusion to work on a single machine. It is not apparent, that the system the system consists out of several loosely coupled workstations. Current network operating systems do not deliver that. In Windows or Linux the user has to define, on which machines the service should run. These systems are slowly maturing in the SMP mode, where multiprocessors share a single memory space. Often these operating systems still lack efficiency, because global locks where used to make processes run consistently on more than on processor. Since locking the whole kernel memory is not necessary in all cases of synchronization, fine grained locking is just about to get introduced. For multicomputers those systems still lack the single-image illusion. They communicate over standard protocols but a centralized control is remained. For this to disappear, a single distributed system call interface and file system has to emerge. For example all kernels need to cooperate and decide where to run a certain process. Decentralized control poses a huge challenge to the software design and the mathematical model is hardly understood.

8.2.5.3 Distributed Computation Paradigm

The important issue in distributed system is to access and manipulate the data in another site of the system because data may be replicated to ensure high availability or may be distributed between many sites to optimize performance. In this section the fundamental modes are discussed to access computations and data in distributed systems.

8.2.5.3.1 Data and Computation Migration

Migration is the movement of data and/or code from one location to another in a distributed environment. Irrespective of granularity of the distributed systems the

ultimate aim of migration is efficient program execution with optimal use of resources and tolerable performance. On a distributed system when a computation attempts to access data from another processor, communication has to be performed to satisfy the reference. For example one of the important factors for efficient program execution on a distributed memory parallel system is the locality of data accesses. If there are many non-local memory accesses it is unlikely that the program would exhibit good speedup. Two mechanisms for accessing remote data are computation migration and data migration, which may be viewed as duals. Computation migration takes the advantage of spatial locality of the objects while data migration takes advantage of the temporal locality of the data objects. Data migration involves moving remote data to the host/processor where a request is made. For example in case of Distributed File System when a client requests access to file that is residing on the server the file is moved to the client's host where it gets cached. In fact more number of file blocks are moved than what is requested and hence subsequent requests are satisfied locally on client's host. Data migration can perform poorly when the size of the data is large. In addition it performs poorly for write shared data because of the communication involved in maintaining consistency: when there are many writes to a replicated datum, it is expensive to ensure consistency. Some of the advantages are that data migration can improve the locality of accesses since after the data is fetched subsequent accesses are local and this model is good when there is high volume of read shared data.

Computation migration involves moving a thread of computation to host/processor where the data is located. It provides a flexible framework for designing distributed systems where the desired non-local computations need not be known in advance at the execution site. But spatial locality of data is not the only reason for doing computation migration. Computation migration can also be used for load balancing and fault recovery. Computation migration is also referred to as mobile computation or mobile agents where code along with its execution state travels on a heterogeneous network until it achieves its goal. This

does not include cases where the code is loaded from the shared disk or from the web (e.g. Java applet). Some of the advantages of this model include:

Efficiency: If repeated interactions with the remote site are required, it can be more effective to send the code to remote site and make it interact locally. This reduces the inter-machine communication cost thus using the network bandwidth efficiently even with high latency. Also it is possible to do load balancing in this model by dynamically assessing the system load and redistributing the work during their lifetime. This helps in achieving better throughput with efficient use of host resources.

Storage: Storage requirements are reduced by not having to store a copy of the program at all sites. In this model loading is done on demand.

Fault Recovery: Unit of execution (which includes code and execution state) can be moved during gradual degradation of performance of the system. If the state of the program can be stored persistently it also possible to move the program even after it crashes.

Flexibility: With this model it is possible for automatic update of the software, as soon it is available at the software vendor irrespective of the number of clients. It could also be very well used in the web push model instead of the traditional pull model.

Simplified Programming: Distributed programming can be simplified by implementing the migration constructs into the programming languages. This provides maximum portability and high transparency to the programmer since explicit distribution of the application into client and server pieces is no longer required. It also allows the programmer to write the programs in a shared-memory style.

The disadvantages of computation migration are that the cost of using it depends on the amount of computation state that must be moved. If the amount of state is large then migration might be fairly expensive. Also when computation migration is used satisfying locality of data and if the data is read shared then data migration might outperform computation migration. Also in heterogeneous distributed environment the differences in the architectures would make

computation migration a difficult task. Both data and computation migration can reduce the communication overhead by making repeated accesses to the local data. As seen data migration involves communication overhead to maintain cache-coherency, which is not involved in computation migration.

8.2.5.3.2 Client-server Computing

Client-server computing is a distributed application architecture that partitions tasks or work loads between service providers (servers) and service requesters, called clients. Often clients and servers operate over a computer network on separate hardware. A server machine is a high-performance host that is running one or more server programs which share its resources with clients. A client does not share any of its resources, but requests a server's content or service function. Clients therefore initiate communication sessions with servers which await incoming requests.

Client-server describes the relationship between two computer programs in which one program, the client program, makes a service request to another, the server program. For example, a web browser is a client program at the user computer that may access information at any web server in the world. To check your bank account from your computer, a web browser client program in your computer forwards your request to a web server program at the bank. That program may in turn forward the request to its own database client program that sends a request to a database server at another bank computer to retrieve your account balance. The balance is returned to the bank database client, which in turn serves it back to the web browser client in your personal computer, which displays the information for you.

Each instance of the client software can send data requests to one or more connected servers. In turn, the servers can accept these requests, process them, and return the requested information to the client. Although this concept can be applied for a variety of reasons to many different kinds of applications, the architecture remains fundamentally the same.

The most basic type of client-server architecture employs only two types of hosts: clients and servers. This type of architecture is sometimes referred to as two-tier.

It allows devices to share files and resources. The two tier architecture means that the client acts as one tier and application in combination with server acts as another tier.

Advantages

- In most cases, client-server architecture enables the roles and responsibilities of a computing system to be distributed among several independent computers that are known to each other only through a network. This creates an additional advantage to this architecture: greater ease of maintenance. For example, it is possible to replace, repair, upgrade, or even relocate a server while its clients remain both unaware and unaffected by that change.
- All data is stored on the servers, which generally have far greater security controls than most clients. Servers can better control access and resources, to guarantee that only those clients with the appropriate permissions may access and change data.
- Since data storage is centralized, updates to that data are far easier to administer than what would be possible under a P2P paradigm. Under a P2P architecture, data updates may need to be distributed and applied to each peer in the network, which is both time-consuming and error-prone, as there can be thousands or even millions of peers.
- Many mature client-server technologies are already available which were designed to ensure security, friendliness of the user interface, and ease of use.
- It functions with multiple different clients of different capabilities.

Disadvantages

- Traffic congestion on the network has been an issue since the inception of the client-server paradigm. As the number of simultaneous client requests to a given server increases, the server can become overloaded.
- The client-server paradigm lacks the robustness of a good P2P network. Under client-server, should a critical server fail, clients' requests cannot be fulfilled. In P2P networks, resources are usually distributed among many nodes. Even if one or more nodes depart and abandon a downloading file, for

example, the remaining nodes should still have the data needed to complete the download.

8.2.5.3.3 Remote Procedure Call (RPC)

It is an Inter-process communication technology that allows a computer program to cause a procedure to execute in another address space (commonly on another computer on a shared network) without the programmer explicitly coding the details for this remote interaction. That is, the programmer would write essentially the same code whether the subroutine is local to the executing program, or remote. When the software in question is written using object-oriented principles, RPC may be referred to as remote invocation or remote method invocation.

RPC is an obvious and popular paradigm for implementing the client-server model of distributed computing. An RPC is initiated by the client sending a request message to a known remote server in order to execute a specified procedure using supplied parameters. A response is returned to the client where the application continues along with its process. While the server is processing the call, the client is blocked (it waits until the server has finished processing before resuming execution).

An important difference between remote procedure calls and local calls is that remote calls can fail because of unpredictable network problems. Also, callers generally must deal with such failures without knowing whether the remote procedure was actually invoked.

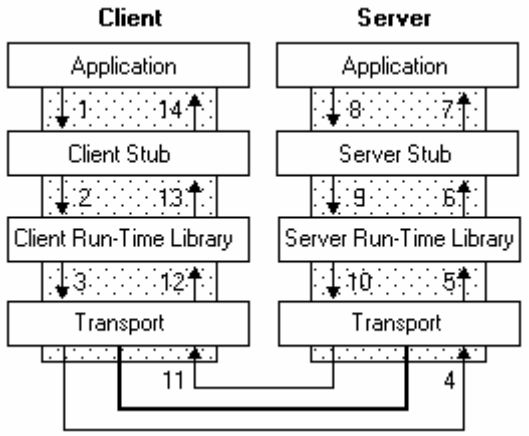
Working of RPC

The RPC tools make it appear to users as though a client directly calls a procedure located in a remote server program. The client and server each have their own address spaces; that is, each has its own memory resource allocated to data used by the procedure. The following figure illustrates the RPC architecture.

As the illustration shows, the client application calls a local stub procedure instead of the actual code implementing the procedure. Stubs are compiled and

linked with the client application. Instead of containing the actual code that implements the remote procedure, the client stub code:

- Retrieves the required parameters from the client address space.
- Translates the parameters as needed into a standard NDR format for transmission over the network.
- Calls functions in the RPC client run-time library to send the request and its parameters to the server.



The server performs the following steps to call the remote procedure.

1. The server RPC run-time library functions accept the request and call the server stub procedure.
2. The server stub retrieves the parameters from the network buffer and converts them from the network transmission format to the format the server needs.
3. The server stub calls the actual procedure on the server.

The remote procedure then runs, possibly generating output parameters and a return value. When the remote procedure is complete, a similar sequence of steps returns the data to the client.

1. The remote procedure returns its data to the server stub.
2. The server stub converts output parameters to the format required for transmission over the network and returns them to the RPC run-time library functions.
3. The server RPC run-time library functions transmit the data on the network to the client computer.

The client completes the process by accepting the data over the network and returning it to the calling function.

1. The client RPC run-time library receives the remote-procedure return values and returns them to the client stub.
2. The client stub converts the data from its NDR to the format used by the client computer. The stub writes data into the client memory and returns the result to the calling program on the client.
3. The calling procedure continues as if the procedure had been called on the same computer.

The run-time libraries are provided in two parts: an import library, which is linked with the application and the RPC run-time library, which is implemented as a dynamic-link library (DLL).

The server application contains calls to the server run-time library functions which register the server's interface and allow the server to accept remote procedure calls. The server application also contains the application-specific remote procedures that are called by the client applications.

8.2.5.3.4 Remote evaluation

Remote evaluation is a general term for any technology that involves the transmission of executable software programs from a client computer to a server computer for subsequent execution at the server. After the program has terminated, the results of its execution are sent back to the client.

Remote evaluation belongs to the family of mobile code technologies. An example for remote evaluation is grid computing: An executable task may be sent to a specific computer in the grid. After the execution has terminated, the result is sent back to the client. The client in turn may have to reassemble the different results of multiple concurrently calculated subtasks into one single result.

8.2.5.4 Important aspects of distributed systems

Five important aspects of true distributed systems are transparency, flexibility, reliability, performance and scalability:

8.2.5.4.1 Transparency

Achieving true transparency requires fooling both the user and the programmer with a single-image of the system. This Goal is motivated by the desire to hide all irrelevant system-dependent details from the user, whenever possible. Shielding the system-dependent information from the users is a trade-off between simplicity and effectiveness. The different types of transparency are:

Access transparency: It is characterized by accessing both local and remote system objects in a uniform way.

Location transparency: In it there is no awareness of object locations. It is sometimes also called name transparency. Location transparency forbids the definition of single machines. Having names specifying a server like /serverhostA/news is not transparent.

Migration transparency: It is the ability to move an object to a different location without changing its name. This is also called location independence. Migration transparency involves that for example processes are free to move from workstation to workstation with their complete state and no implication for the program. Local resources need to be proxied to the new location in this case.

Concurrency transparency: It allows the sharing of objects without interference.

Replication transparency: It may be defined as consistency of multiple instances (or partitioning) of files and data.

Parallelism transparency: It permits parallel activities without users knowing how, when and where. Parallel transparency even requires that the programmer should not be bothered on how to make his algorithms to run in parallel. Achieving this is the holy grail of distributed systems and no current existing systems even touch this.

Failure transparency: It makes the system fault tolerance.

Performance transparency: It attempts to achieve a consistent and predictable performance level even with changes of the system structure or load distribution.

Size transparency: It permits modularity and scalability.

Revision transparency: It results into vertical growth of the system.

8.2.5.4.2 Flexibility

Flexibility is very important for a distributed system since changes in this environment is inherent and not all design decisions yield to a possible system. It should be easy to reverse decisions without losing previous work. Since nobody can argue against flexibility it leaves to define what it means for operating systems. Two concepts exist today. Micro kernels are mainly responsible for sending messages to the user level processes. On the other hand the monolithic kernel tries to solve user program requirements under the security shield of the kernel space. The performance of micro-kernels may be worse since more context switches are required. Also the task of providing security is harder in this system, since the security has to be valid in the user space and can not be hidden in the kernel space. But once security is achieved in micro kernel, it is no problem to extend that to multiple computer. Flexibility includes the friendliness of the system and the freedom of the user in using the system - ease of use of the system interface and the ability to relate the computation processes in the user's problem domain to the system. The object-oriented strategy is a commonly used strategy in achieving this goal. From the system's view flexibility is the system's ability to evolve and migrate - modularity, scalability, portability and interoperability.

8.2.5.4.3 Reliability

The goal of reliability is one of the main reasons for a distributed system. Availability of services is a main aspect. This can be achieved by decoupling dependant services in the design and redundancy. As soon as one machine fails another takes over the task. These services and their usage should be fault-tolerant. The failure of one composed should not bring down the entire program.

8.2.5.4.4 Performance

Transparency and reliability are very expensive in terms of performance. Adding or multiplying integers remotely is never reasonable, since the message overhead eliminates the benefit of doing this remotely. In some cases it is impossible to predict, whether a remote operation is a gain in performance, because the computation may depend on external events.

Efficiency is more complex in distributed systems than in centralized systems due to the effect of communication delays. With respect to system load distribution, problems such as bottlenecks and congestion either in the physical networks or software components must be addressed. Computation speed and system throughput can be enhanced through distributed processing and load sharing if the communication system is carefully designed.

8.2.5.4.5 Scalability

Considerable amount of design effort has to deal on how to extend a system. Centralized structures like a database which serves all incoming request may soon turn out to be a bottleneck for the whole system. Congestions in network traffic prevent the system to grow further in size. A good example of a scalable system is the DNS. Here database entries and requests to the root servers are highly decentralized in a hierarchical structure. This scalability is paid with the price of very slow changes of entries. It sometimes takes days for changes to propagate, but even Denial of Service attacks have proven to be hard to conduct against the DNS.

8.2.5.4.5 Consistency

It is more difficult to achieve in a distributed system due to the lack of global information, potential replication and partitioning of data, the possibility of component failures and the complexity of interaction among modules. The system must be capable of maintaining its integrity with proper concurrency control mechanisms and failure handling and recovery procedures.

8.2.5.4.7 Robustness

Failures (in communication links, processing nodes and client/server processes) are more frequent than in a centralized single computer system. Meaning of robustness is (a) fault tolerance: ability to reinitialize itself to a consistent state with only some possible degradation of its performance, and (b) security

8.3 Summary

Network operating systems, abbreviated as NOS, are characterized by information exchange divided and implemented from communication sub-network to transport services and by inclusion of a transport layer and the support for

network applications like remote login, file transfer, messaging, network browsing, and remote execution. NOS include special functions for connecting computers and devices into a local-area network (LAN). Some operating systems, such as UNIX and the Mac OS, have networking functions built in. The term network operating system, however, is generally reserved for software that enhances a basic operating system by adding networking features.

Distributed operating systems are characterized by sharing of resources and coordination of distributed activities in networked environments but the key distinction between a network Operating System and a distributed Operating System is the concept of transparency which hides all the unnecessary details from the user. Flexibility, reliability, and scalability are some other important aspects of distributed operating systems. Data migration, computation migration, Remote Procedure Call, Remote evaluation, and client-server computing are the fundamental modes which are used to manipulate the data in distributed systems.

8.4 Keywords

Data migration: It is moving of remote data to the host/processor where a request is made.

Computation migration: It is moving of a thread of computation to host/processor where the data is located.

Client-server computing: It is a distributed application architecture that partitions tasks between service providers (servers) and service requesters, called clients.

Remote Procedure Call: It is an Inter-process communication technology that allows a computer program to cause a procedure to execute on another computer on a shared network without the programmer explicitly coding the details for this remote interaction.

Remote evaluation: It is the transmission of executable software programs from a client computer to a server computer for subsequent execution at the server. After the program has terminated, the results of its execution are sent back to the client.

8.5 Self Assessment Questions (SAQ)

1. What do you understand by Remote Procedure Call (RPC)? Write a detailed note on its implementation.
2. What do you understand by distributed operating system? What are the factors motivating the development of distributed operating systems.
3. What are the important differences between network operating system and distributed operating system?
4. Write a detailed note on the transparency aspect of distributed operating systems.
5. What is client-server computing? Explain.
6. What are the different architecture models of distributed operating systems? Explain.
7. Write short notes on following:
   (a) Data and Computation migration
   (b) Remote evaluation
   (c) Network topologies
   (d) LAN and WAN

8.6 Suggested Readings / Reference Material

6. Operating System Concepts, 5th Edition, Silberschatz A., Galvin P.B., John Wiley & Sons.
7. Systems Programming & Operating Systems, 2nd Revised Edition, Dhamdhere D.M., Tata McGraw Hill Publishing Company Ltd., New Delhi.
8. Operating Systems, Madnick S.E., Donovan J.T., Tata McGraw Hill Publishing Company Ltd., New Delhi.
9. Operating Systems-A Modern Perspective, Gary Nutt, Pearson Education Asia, 2000.
10. Operating Systems, Harris J.A., Tata McGraw Hill Publishing Company Ltd., New Delhi, 2002.

Lesson Number: 9	***Writers: Dr. Rakesh Kumar***
UNIX Operating System	***Vetter: Dr. Pradeep Bhatia***

9.0 Objectives

The objective of this lesson is

(A) To give an overview of the important features of UNIX operating system to the students.

(B) To make the students familiar with some important UNIX commands.

9.1 Introduction

UNIX is written in a high-level language giving it the benefit of machine independence, portability, understandability, and modifiability. Multi-tasking and multi-user are the two most important characteristics of UNIX helping it in gaining widespread acceptance among a large variety of users. It was the first operating system to bring in the concept of hierarchical file structure. It uses a uniform format for files called the byte stream making the application programs to be written easily. UNIX treats every file as a stream of bytes so the user can manipulate his file in the manner he wants. It provides primitives that allow more complex and complicated programs to be built from the simpler ones. It provides very simple user interface both character-based and graphical based. It hides the machine architecture from the user. This helps the programmer to write different programs that can be made to run on different hardware configurations. It provides a simple, uniform interface to peripheral devices.

9.2 Presentation of contents

9.2.1 Versions

9.2.2 UNIX Architecture

9.2.3 Features of UNIX

9.2.1 Versions

Some popular versions of UNIX are AIX IBM, XENIX, ULTRIX, Sun OS, and BSD. The original version of UNIX came in the late 1960's designed by Ken Thompson at AT&T Bell Laboratories. At that time, Bell Labs were busy in designing an operating system called Multics with an objective to provide a very sophisticated and complex multi-user operating system that had support for many advanced features. However, Multics failed because the state of art provided by it at that time was too complex. So Ken Thompson then started working on

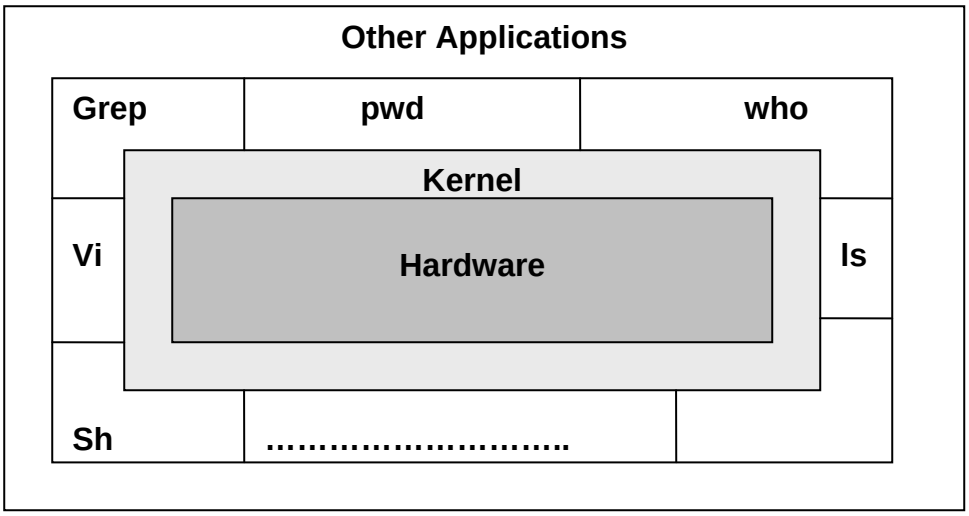
a simpler project and he named it UNIX. Dennis Ritchie rewrote the source code of UNIX operating system in C language.

By the year 1977, UNIX I system found its major contribution in the telephone companies, providing a good environment for program development, network transaction services and real time services. In the year 1977, the UNIX system was first ported from a PDP to a non-PDP machine. As the popularity of UNIX grew, many other companies came out with their own versions of UNIX and ported it onto other new machines. From the year 1977 to 1982, Bell Laboratories combined many AT & T variants into a single system and gave it a name UNIX System III. Bell Laboratories in this version brought out many new features and advancements. It was given the name UNIX System V. The people at University of California at Berkeley developed a variant to the UNIX System. Its recent version is called 4.3 BSD for VAX machines.

9.2.2 UNIX Architecture

The high level architecture of the UNIX system is shown in Figure 9.1.

Figure 9.1 System Architecture of UNIX



The UNIX system is organized as a set of layers. The Kernel surrounds the hardware. The user programs are independent of the hardware on which they are running. The programs such as the shell and editors interact with the Kernel by invoking a well-defined set of system calls. The system calls get various actions

done from the Kernel for the calling program. They interchange data between the Kernel and the program. There are many other programs in this layer which from a part of the standard system configurations. These programs are known as commands. There are several other user created programs present in the same layer. The outer-most layer contains other application programs, which can be built on top of lower level programs. For instance, the C compiler appears in the outermost layer of the figure. It invokes a C preprocessor, compiler, assembler and link loader. These are all separate lower level programs. The programming style offered by the UNIX system helps us to fulfill a task by combining the existing programs.

9.2.3 Features of UNIX

The popularity of UNIX is due to the following reasons:

Portability: UNIX is a portable operating system i.e. it can run successfully on all types of computers. The reason of this is that it is written in a high-level language. PCs, Macintoshes, Workstations, Minicomputers, Super Computers and Mainframes run the UNIX operating system with equal ease.

Machine Independent: The UNIX system does not expose the machine architecture to the user. Thus, it becomes very easy to write applications that can run on micros, minis or mainframes.

Multi-user Capability: UNIX is a multi-user system in which the same computer resources like hard disk; memory etc can be used or accessed by many users simultaneously. Each user is given a terminal. Each terminal is an input and an output device for the user. All the terminals are connected to the main computer. So, a user sitting at any terminal can not only uses the data or the software of the main computer but also the peripherals like printers attached to it. The main computer is called the server or the console. The number of terminals that can be connected to the server depends upon the number of parts present in the controller card.

Multitasking Capability: UNIX has the facility to carry out more than one job at the same time. Multitasking is achieved by dividing the CPU time in the order of milliseconds/microseconds for execution between all the jobs that are being carried

out. Each job is carried out according to its priority number. It gives the impression that the tasks are being carried out simultaneously.

Software Development Tools: UNIX offers an excellent environment for developing new software. It provides a variety of tools ranging from editing a program to maintenance of software. It exploits the power of hardware to the maximum extent of effectiveness and efficiency.

Built-in Networking: UNIX has got built in networking support with a large number of programs and utilities. It also offers an excellent media for communication with other users. The users have the liberty of exchanging mail, data, programs, etc. You can send your data at any place irrespective of the distance over a computer network.

Security: UNIX enforces security at three levels.

(a) Each user is assigned a login name and a password. So, only the valid users can have access to the files and directories.

(b) Each file is bound around permissions (read, write, execute). The file permissions decide who can read/ modify/ execute a particular file. The permissions once decided for a file can also be changed from time to time.

(c) Then file encryption comes into picture. It encodes file in a format that cannot be very easily read. So, if anybody happens to open file, even then he will not be able to read the text of the file. However, you can decode the file for reading its contents.

9.2.4 Implementation of Operating System Functions

UNIX operating system performs following designated functions:

(a) Process management: creating, destroying & manipulating processes.

(b) Memory management: allocating, de-allocating and manipulating memory.

(c) Input/Output: communicating and controlling I/O device and file system.

(d) Miscellaneous: Network functions etc.

The UNIX System V offers somewhere around 64 system calls, which carry very simple options with them. So, it becomes easy to make use of these system calls. The body of the Kernel is formed by the set of system calls and the internal

algorithms that implement them. Kernel provides all the services to the application programs in the UNIX system. In UNIX, the programs do not have any knowledge of the internal format in which the Kernel stores file data.

9.2.4.1 Process management functions

The behavior of a UNIX process is defined by its text segment, data segment and stack segment as shown in Figure 9.2.

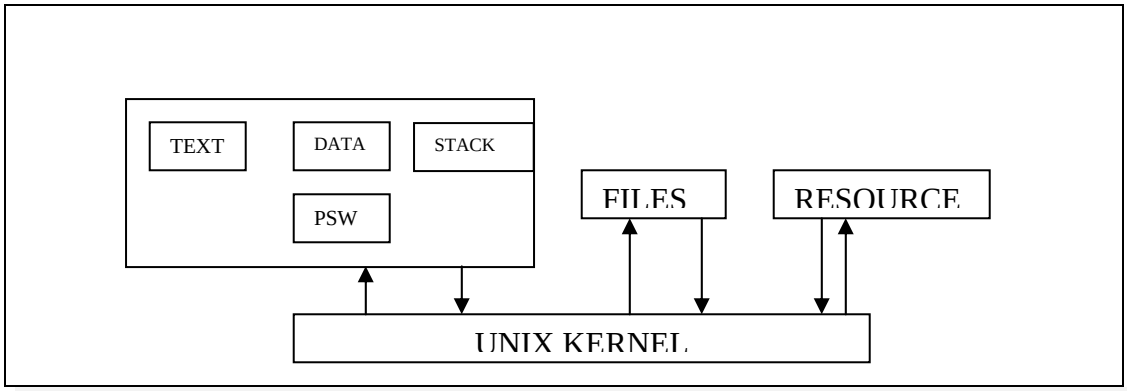


Figure 9.2: A Process in UNIX

The text segment contains the compiled object instructions, the data segment contains static variables, and the stack segment holds the runtime stack used to store temporary variables. A set of source file that is compiled and linked-into an executable form is stored in a file with the default name of a.out. If the program references statically define data, such as C static variables, a template for the data segment is maintained in the executable file. The data segment will be created and initialized to contain values and space for variables when the executable file is loaded and executed. The stack segment is used to allocate storage for dynamic elements of the program, such as automatic C variables that are created when they come into scope and are destroyed when pass out of scope.

The compiler and linker create the executable file. These utilities do not define a process; they define only the program text and a template for the data component that the process will use when it executes the program. When the loader loads a program into the computer's memory, the system creates appropriate data and stack segments, called a process.

A process has a unique process identifier (PID), a pointer to a table of process descriptors used by the UNIX Operating System kernel to reference the process's

descriptor. Whenever one process references another process in a system call, it provides the pointer of the target process. The UNIX pa command lists each pm associated with the user executing the command. The pm of each process appears as a field in the descriptor of each process.

UNIX command for creating a new process is the fork system call. Whenever a process calls fork, a child process is created with its descriptor, including its own copies of the parent's program text, data, and segments, and access to all open file descriptors (in the kernel). The child and parent processes execute in their own separate address spaces. This means that even though they have access to the same information, both the child and its parent each reference their own copy of the data. No part of the address space of either process is shared.

Hence, the parent and child cannot communicate by referencing variables stored at the same address in their respective address space. UNIX systems also provide several forms of the execve system call to enable a process to reload its address space with a different program:

execve (char *path, char *argv[], char *envp[]);

This system call causes the load module stored in the file at path to replace the program currently being executed by the process. After execve has completed executing, the program that called it is no longer loaded. Hence, there is no notion of returning from an execve call, since the calling program is no longer loaded in memory. When the new program is started, it is passed the argument list, argv, and the process uses a new set of environment variables, envp.

UNIX also provides a system call, wait (and a variant, waitpid), to enable a parent process to detect when one of its child processes terminates. Details of the terminating child's status may be either returned to the parent via a value parameter passed to wait or ignored by the parent. The waitpid allows the parent to wait for a particular child process (based on its PID) to terminate, while the wait command does not discriminate among child processes. When a process exits, its resources, including the kernel process descriptor, are released. The operating system signals the parent that the child has died, but it will not release the process descriptor until

the parent has received the signal. The parent executes the wait call to acknowledge the termination of a child process.

Following are the rest of UNIX system calls related to process management:

- acct enable/disable process accounting
- alarm set a process alarm clock
- exit terminate a process
- fork create a new process
- getpid get process, process group and parent process ID
- getuid get real user, effective user real group and effective group ID
- kill send a signal to a process or group of processes
- msgctl message control operation msgop message operation
- nice change priority of a process pause suspend until
- pipe create an inter-process channel
- profil execution time profile ptrace process trace
- semctl semaphor control operations
- semget get set of semaphor
- semop semaphor operations
- setpgrp: set process group ID
- setuid set group and user ID
- signal specify what to do when a signal is received
- stime set time
- sync update super block time get time
- times get process and child process times
- ulimit get user upper limits
- uname get the name of the current operating system
- ulink remove directory entry
- Wait wait for the child process to stop or terminate

9.2.4.2 Memory Management

Kernel resides in the main memory so long as computer is operational. When a program is compiled, a set of addresses is generated in the program by the

compiler. These represent addresses of variables or addresses of instructions such as functions. The addresses generated by the compiler are for a virtual machine. The addresses are not absolute in terms of memory addresses where they will be loaded eventually. It assumes that no other program will be executing concurrently.
However, when you run the program the Kernel allocates some space to it in the main memory. But the virtual addresses generated by the compiler might not resemble the physical addresses occupied in the machine. Then the Kernel maps the compiler-generated address with the physical machine addresses.
UNIX divides the available memory into system memory and application memory. It loads itself into system memory and creates data structures it will use in its operation into this area of memory. The application memory area contains the user's programs. The application memory area is divided into global stack, local stack and heap. The global and static type of variables, functions etc. are assigned space in these memory areas. UNIX provides system calls to affect loading and unloading of programs into and out of the processes. Internally UNIX uses paging with segmentation methods to manage memory. In addition to these primitive operations, UNIX provides library functions like malloc, for allocating memory to a process and objects dynamically. Other memory related system calls are:

- brk change data segment space allocation
- shmop shared memory operations
- shmctl shared memory control operation
- shmget get shared memory segment
- plock lock process, text or data memory addresses,
- msgget get message queue
    - UNIX uses non-contiguous memory allocation with paging hence memory allocators use an integral number of pages.
    - **McKusick_Karels allocator**
    - This is used in UNIX 4.4 BSD and uses an integral number of pages. In it each page is divided into blocks of equal size and stores this size information along the logical address of the page so size need not to be stored in each free block. While freeing a block the free lit to which the block should be

added is found by finding the address of the page to which the block belongs. So there is no need of header element. Unlike the buddy system this allocator does not coalesce adjacent free blocks. In stead it makes the page as free when all the blocks in it become free.

- **Lazy Buddy Allocator**
- In buddy system, one or more splitting or coalescing takes place whenever there is some allocation or release of memory block takes place respectively. Some of these splitting/coalescing could be avoided because a coalesced block may split in future again resulting in the improvement of performance. Unix 5.4 lazy buddy allocator. Here the states of a block of class are characterized as lazy, reclaiming, and accelerated. In the lazy state allocation and release of a blocks of a class occur with matching frequencies. Hence coalescing may lead to split so both can be avoided by delaying the coalescing. In the reclaiming state, releases occur at a faster rate than allocation so it is a good idea to coalesce at every release. In the accelerated state, releases occur at much faster rate than allocation, so allocator not only tries to coalesce the block being released but also those blocks which have been released in the past.
- UNIX virtual memory
- It differentiates between three kinds of pages, (i) Resident page – that exists in memory, (ii) Un-accessed page – that has not been accessed even once during execution of the process, and (iii) swapped out – this page exists in the swap space.

9.2.4.3 Device and File functions

For each device it has device drivers for low-level communication. UNIX treats every device the same way as it treats the files. Device drivers are intended to be accessed by user space code. If an application accesses a driver, it uses one of two standardized interfaces: the block device interface or the character device interface. Both interfaces provide a fixed set of functions to the user programs.

When a user program calls the driver, it performs a system call. The kernel searches the entry point for the device in the block or character in direct reference table (the

jump table) and then calls the entry point. The exact semantics of each function depends on the nature of the device and the intent of the driver design. Hence, the function names suggest only a purpose for each. The logical contents of the jump table are kept in the file system in the dev directory.

A UNIX driver has three parts:

- Code to initiate derive operations
- Device interrupt handlers

The initialisation code is run when the system is booted or started first time. It tests for the physical presence of respective devices and then initializes them. The API implements functions for a subset of the entry points. This part of the code also provides information to the kernel as to which functions are implemented. The system interrupt handler that corresponds to the physical device causing the interrupt calls the device interrupt handler.

System administrators are responsible for installing devices and drivers. The information necessary to install a driver can be incorporated into a configuration file by the administrator and then processed by the configuration builder tool /etc/conf.

UNIX has system calls to effect I/O manipulation. Some of them are:

- write - write on a file
- utime - set file access and modification times
- fstat - get file system statistics
- ulink - remove directory entry
- umount - unmount a file system
- umask - set and get file creation mask
- stat - get file status
- read - read from a file
- open - open for reading or writing h
- mount - mount a file system
- mknod - make a directory or special or ordinary file
- lseek - move read/write pointer link link to a file

- bfcntl - file control
- nexec - execute a file
- lytodup - duplicate an open file descriptor
- dbycreat - create a new file or rewrite an existing one
- close - close a file descriptor let of chroot change to root
- chown - change owner or group of file entry
- chmod - change mode of file
- chdir - change directory device access determine accessibility of a file

9.2.5 UNIX Kernel

The services provided by the Kernel are given below:

1. It controls the fate and state of various processes such as their creation, termination and I/O suspension.
2. The Kernel allocates main memory for an executing process. The Kernel allows the processes to share portions of their address space. It keeps the private space of processes secure and does not allow tampering from other processes. However, if the free memory is low with the system, then the Kernel frees out some memory by writing a process temporarily to secondary memory. In case the Kernel writes all the processes to the secondary memory, it is called a swapping system. However, if only the pages of memory are written onto the secondary memory, then it is called the paging system.
3. The Kernel schedules processes for execution on the CPU. The time-sharing concept allows the processes to share the CPU. When the time of a process has finished, the Kernel suspends it and puts some other ready process for execution in the CPU. It is again the work of the Kernel to reschedule the suspended process.
4. Kernel permits different processes to make use of the peripheral devices such as terminals, tape drives, disk drives & network devices as & when requested.
5. The Kernel allocates the secondary memory for efficient storage and retrieval of user data. The Kernel allocates secondary storage for user files, organizes the file system in a well-planned manner and provides security to user files from illegal access.

6. The services provided by the Kernel are absolutely transparent to the user. For instance, the Kernel formats the data present in a file for internal storage. However, it hides the internal format from user processes. Similarly, it makes a distinction between the regular file and a device but hides the distinction from user processes. Finally, the Kernel provides the services so that the user level processes can support the services they must provide. For instance, the Kernel provides the services that the shell requires to act as a command interpreter. Therefore, the Kernel allows the shell to read terminal input, to create pipes and to redirect I/O.

9.2.5.1 Assumptions about Hardware

Whenever the user on the UNIX system executes a process, it is divided into two levels: User level and Kernel level. So, as and when a process executes a system call, the execution mode of the process changes from the user mode to Kernel mode. The Kernel tries to process the requests made by the user. It returns an error message if the process fails. However, if no requests are given to the operating system to service, even then the operating system keeps itself busy with other operations such as handling interrupts, scheduling processes, managing memory and so on. The main differences between the user mode and the Kernel mode are given below:

1. Process in a user mode can access their own instructions and data but they cannot access the instructions and data of the Kernel. But all the processes present in the Kernel can have the access to both the Kernel and the user addresses.
2. Some machine instructions give an error message when executed in user mode. For instance, a machine may contain an instruction that manipulates the processor status register. This instruction is not allowed to be executed in user mode. Processes executing in user mode should not have this capability otherwise they may corrupt the kernel loaded.

The following three situations result in switching to kernel mode from user mode of operation:

1. The scheduler allocates a user process a slice of time (about 0.1 second) and then system clock interrupts. This entails storage of the currently running process status and selecting another runnable process to execute. This switching is done in kernel mode. A point that ought to be noted is: on being switched the current process's priority is re-evaluated (usually lowered). The Unix priorities are ordered in decreasing order as follows:
    - HW errors
    - Clock interrupt
    - Disk I/O
    - Keyboard
    - SW traps and interrupts
2. Services are provided by kernel by switching to the kernel mode. So if a user program needs a service (such as print service, or access to another file for some data) the operation switches to the kernel mode. If the user is seeking a peripheral transfer like reading a data from keyboard, the scheduler puts the currently running process to “sleep" mode.
3. Suppose a user process had sought a data and the peripheral is now ready to provide the data, then the process interrupts. The hardware interrupts are handled by switching to the kernel mode. In other words, the kernel acts as the via-media between all the processes and the hardware.

9.2.5.2 Interrupts and Exceptions

The UNIX system allows devices such as I/O peripherals or the system clock to interrupt the CPU abruptly. Whenever the Kernel receives the interrupt, it saves the current work it is doing and services the interrupt. After the interrupt is processed, the Kernel resumes the interrupted work and proceeds as if nothing had happened. The hardware gives a priority weightage according to the order in which the interrupts should be handled. Thus, when the Kernel looks into an interrupt, it keeps the lower priority interrupts waiting and services the higher priority interrupts.

The term exception is different from the term interrupt. An exception is a condition in which a process causes an unexpected event. For example: dividing a number by zero, illegal address, out of memory, etc. Exceptions occur in the middle of the execution of an instruction and are the similar to interrupts. The system tries to start the instruction again after handling the exception. However, interrupts are considered to happen between the executions of two instructions. The system continues working on the next instruction after servicing the interrupt.

9.2.5.3 Processor Execution Levels

Sometimes, the Kernel must stop the interrupt from occurring during critical activity preventing data corruption. For instance, the Kernel might not want to handle an interrupt when it is working with linked lists because handling the interrupt at this point of time might lead to corruption of pointers. Therefore, a better technique has been worked out. The processor execution levels can be set with the help of certain instructions. If you set the processor execution level to certain value, then it can keep away the interrupt from that level and lower levels. It will only allow the high level interrupts to disturb the process.

9.2.6 File System and Internal Structure of Files

Kernel does not impose any structure on files, and no meaning is attached to its contents - the meaning of bytes depends solely on the program that interprets the file. This is not true of just disc files but of peripherals devices as well. Magnetic tapes, mail messages, character typed on the keyboard, line printer output, data flowing in pipes - each of these is just a sequence of bytes as far as the system and the programs in it are concerned.

Files are organized in tree-structured directories. Directories are themselves files that contain information on how to find other files. A path name to a file is a text string that identifies a file by specifying a path through the directory structure to the file. Syntactically it contains of individual file name elements separated by the slash character.

The UNIX file system supports two main objects: files and directories. Directories are nothing but files, which have a special format.

9.2.6.1 Representation of Data in a File

All the data entered by the user is kept in files. Internally the data blocks take up most of the data that has been put in files. Each block on the disk is addressable by a number. Associated with each file in UNIX is a little table called inode, which contains the table of contents to locate a file's data on disk. The table of contents consists of a set of disk block numbers. An inode maintains the attributes of a file, including the layout of its data on disk. Disk inodes consists of the following fields:

- Last modification date
- Last access date
- Time the file last read
- Last inode modification
- Time the file was last modified
- Reference count
- Block reference pointer and indirect pointer to blocks in the file

The data on a file is not stored in a contiguous section of the disk. The reason behind is that the Kernel will have to allocate and reserve continuous space in the file system before allowing operations that would increase the file size. For instance, let us suppose that there are three files A, B and C. Each file consists of 10 blocks of storage and supposes the system allocated storage for the three files contiguously as shown in Figure 9.3.

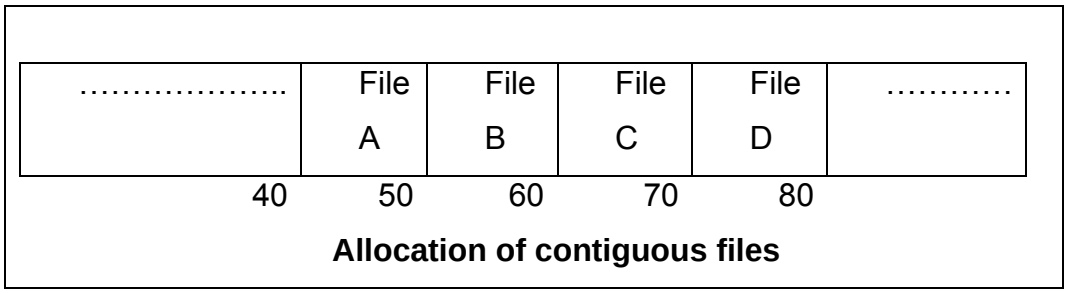


Figure 9.3

However, if the user now wishes to add 5 blocks of data in the file B, then the Kernel will have to copy the file to such a place where a file of 15 blocks can be accommodated. Moreover, the previously occupied disk block by file B's data can only be used in a case where the files have data less than 10 blocks.

The Kernel allocates the file space of one block at a time. This allows the data to be spread throughout the file system. In this case, locating the data of a file becomes a

complicated process. If a block contains 10K bytes, then such a file would need an index of 100 block numbers and as the block of 100K bytes would need an index of 1000 block numbers. Thus, the size of the inode would keep varying according to the size of the file.

9.2.6.2 Directories

The directories are files that give the file system a hierarchical structure. In a directory the data is put in a sequence of entries. Each such entry contains an inode number and the name of a file present in the directory .The pathname is a null terminated character string. The pathname is divided into separate parts by the / (slash) character. Each component of the pathname should hold the name of a directory. However, the very last component can be a non-directory file. The component names can have a maximum of 14 characters, with a 2 byte entry for the inode number, the size of a directory entry is 16 bytes. Each directory contains the file names dot and dot-dot. The inode numbers of these directories are those of the directory and its parent directory respectively. The inode number of "." in "\etc" directory is present at offset 0 in the file and its value is 83. The inode number of ".." is present at the offset 16 and its value is 2. Any directory entry can also be kept empty. Its inode number is indicated by 0.

The data stored by the Kernel for a directory is similar to the data stored for an ordinary file. For the directory also the Kernel makes use of the inode structure and direct and indirect blocks. The access permission of a directory has the following meaning: the read permission allows a process to read a directory. Write permission allows a process to create new directory entries and remove the old directory entries. It accounts for altering the contents of a directory. The execute permission allows a process to search the directory for a filename.

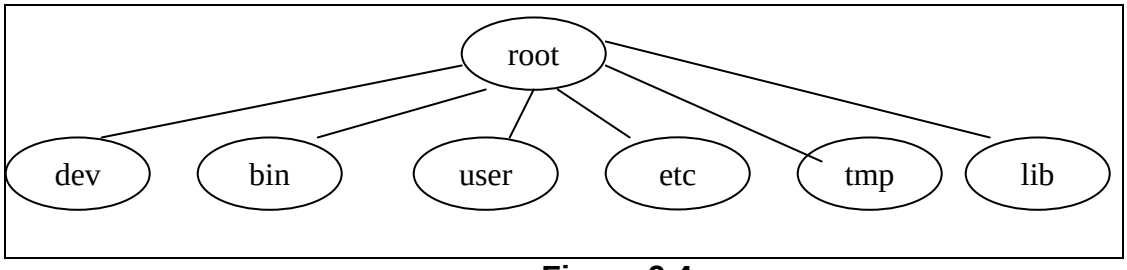


Figure 9.4

Figure 9.4 shows a typical UNIX File System. The file system is organized as a tree with a single root node called the root (written "/ "); every non-leaf node of the file system structure is a directory, and leaf nodes of the tree are either directories or regular files or special devices.

- The /bin directory contains the executable files for most UNIX commands.
- The /etc directory contains other additional commands that related to system maintenance and administration. It also contains several files, which store the relevant information about the users of the system, the terminals and devices connected to the system.
- The /lib directory contains all the library functions provided by UNIX for the programmers.
- The /dev directory stores files that are related to the devices. UNIX has a file associated with each of the I/O devices.
- The /user directory is created for each user to have a private work area where the user can store his files. This directory can be given any name. Here it is named as "user".
- The /tmp directory is the directory into which temporary files are kept. The files stored in this directory are deleted as soon as the system is shutdown and restarted.

Create, open, read, write are system calls, which are used for basic file manipulation. The create system call; given a path name creates an empty file. An existing file opened by the open system call, which takes a path name and a node and returns a small descriptor which may then be passed to a read or write system call to perform data transfer to or from the file.

A file descriptor is an index into a small table of open files for this process. Descriptors start at 0 and seldom get higher than 6 or 7 for typical programs, depending on the maximum number of simultaneously open files. Each read or write updates the current offset into the file, which is associated with file table entry and is used to determine the position in the field for the next read or write.

9.2.6.3 Blocks and Fragments

Most of the file system is taken up by data blocks, which contain whatever the users have put in their files. The hardware disk sector is usually 512 bytes. A block size larger than 512 bytes is desirable for a speed. However, because UNIX file system usually contain a very large number of small files, much larger blocks would cause excessive internal fragmentation. That is why the earlier 4.1 BSD file system was limited to 1024-byte block. The 4.2 BSD solution is to use two block sizes for files which have no indirect blocks: all the blocks of the file are large block size except the last. The last block is an appropriate multiple of a smaller fragment size to fill out the file. Thus, a file of size 18000 bytes would have two 8K blocks and one 2K block fragment.

The block and fragment sizes are set during the file creation according to the intended use of the file system: if many small files are expected, the fragment size should be small; if repeated transfer of large files are expected, the basic block size should be large.

9.2.7 UNIX Shell

A shell is the user-interface to the UNIX. A shell could use many different strategies to execute a user's computation. The approach used in modern shells is to create a new process to execute new computation. For example, if a user decides to compile a program, the process interacting with the user creates a new child process to carry out the compilation task to execute the compiler program. The initial process (the OS) can use this same technique when it decides to service a new interactive user in a timesharing environment. That is, when the user attempts to establish an interactive session, the Operating System treats this as a new computation. It awakens a previously created process for the login port or creates a new process to handle the interaction with the user.

This idea of creating a new process to execute a computation is a very important characteristic. When the original process decides to execute a new computation, it protects itself from any fatal errors that might arise during that execution. If it did not use a child process to execute the command, a chain of fatal errors could cause the initial process to fail, thus crashing the entire system.

The Bourne shell and others accept a command line from the user, parse the command line, and then invoke the Operating System to run the specified command with the specified

arguments. When a user passes a command line to the shell, it is interpreted as a request to execute a program in the specified file - even if the file contains a program that the user wrote.

For example, you could write a C program in a file named main.c, then compile and execute it with shell commands like

```
$ cc main.c
$ a.out
```

The shell finds the cc command (the C compiler) in the /bin directory, and then passes it the string "main.c" when it creates a child process to execute the cc program. The C compiler, by default, translates the C program that is stored in main.c, then writes the resulting executable program into a file named a.out in the current directory. In the second command, the command line is just the name of the file to be executed, a.out. The shell finds the a.out file in the current directory and then executes it.

Consider the detailed steps that a shell must take to accomplish its job:

- Printing a prompt: There is a default prompt string, e.g., the single character string "%", "#", ">" or other. When the shell is started, it can look up the name of the machine on which it is running, and prepare this string name to the standard prompt character.
- Once the prompt string is determined, the shell prints it to screen whenever it is ready to accept a command line.
- Getting the command line: To get a command line, the shell performs a blocking read operation so that the process that executes the shell will be blocked until the user types a command line in response to the prompt. When the command has been provided by the user and terminated with a NEWLINE character, the command line string is returned to the shell.
- Parsing the command: The parser begins at the left side of the command line and scans until it sees a white space character. The first word is treated as the command name, and subsequent words are treated as parameter string.
- Finding the file: The shell provides a set of environment variables for each user- this variable is first defined in the user's login file, though it can be modified at

any time with the set command. The PATH environment variable is an ordered list of absolute pathnames that specifies where the shell should search for command files. If the login file has a line such as set path=(.:/bin:/usr/bin)

The shell will first look in the current directory (since the first pathname is "." for the current directory), then in /bin, and finally in /usr/bin. If there is no file with the same name as the command in any of the specified directories, the shell responds to the user that it is unable to find the command.

- Preparing the parameters: The shell simply passes the string parameters to the command as the argv array of pointers to strings.
- Executing the command: Finally the shell must execute the binary object program in the specified file. UNIX shells have always been designed to protect the original process from crashing when it executes a program. That is, since a command can be any executable file, the process that is executing the shell must protect itself in case the executable file has a fatal error in it. Somehow, the shell wants to "launch" the executable so that even if the executable contains a fatal error (which destroys the process executing it), the shell will remain unharmed.

The Bourne shell uses multiple processes to accomplish what the UNIX-style system calls fork, execve, and wait. This system call creates a new process, which is a copy of the calling process except that it has its own process identification (with the correct relationships to the other processes) and its own pointers to shared kernel entities such as file descriptors. After fork has been called, two processes will execute the next statement after the fork in their own address spaces - the parent and the child. If the call succeeds in the parent process, fork returns the process identification of the newly created child process, and in the child process, fork() returns a zero value.

execve. This system call is used to change the program that the process is currently executing. It has the form execve (char *path, char *argv[], char *envp[]. The path argument is the pathname of a file that contains the new program to be executed. The argv array is a list of parameter strings, and the envp array is a list of environment variable strings and values that should be used when the process begins executing the new program. When a process encounters the execve system

call, the next instruction it executes will be the one at the entry point of the new executable file. This means that the kernel performs a considerable amount of work in this system call. It must find the new executable file, load it into the address space currently being used by the calling process (overwriting area and discarding the previous program), set the argv array and environment variables for the new program execution, then start the process executing at the new program's entry point.

wait. A process uses this system call to block itself until the kernel signals the process to execute again. For example, because one of its children processes has terminated. When the wait call returns as a result of a child process terminating, the status of the terminated child is returned as a parameter to the calling process.

9.2.8 User Interaction with UNIX Operating System

To interact with UNIX, the first step is login process in which we use a name and password initially assigned by the system administrator.

9.2.8.1 Steps to Login

Logging in is a procedure that tells the UNIX System who you are. On power on you would find the Login prompt on the screen. Each user on the UNIX system is assigned an account name (combination of eight or less characters), which identifies him as a unique user. Once the login name is entered, UNIX prompts you to enter a password. If you give either the login name or the password wrong, then UNIX denies you the permission to access its resources & shows an error message on the screen. Once you have successfully logged on, you will find $ prompt (It is the default prompt in Korn or Bourne shells). Now it is ready to accept commands from the user.

When you are done working on your system and decide to leave, you can log off the system by typing the following command in Bourne or Korn shell:

$ exit

However, if you are working on C shell, you can give another command to log off.

$ logout

9.2.8.2 Changing your Password

To change your password, issue the 'passwd' command at the UNIX prompt.

Syntax: passwd [user-name] Options

Only the super user can use the options given below:

-d Deletes your password

-x days This sets the maximum number of days that the password will be date active. After the specified number of days you will be required to give a new password.

-n days. This sets the minimum number of days the password has to be active, before it can be changed.

-s This gives you the status of the user's password.

UNIX also offers a variety of tools to maintain security .One such tool is the 'lock' command that locks your keyboard till the time you enter a valid password.

9.2.8.3 UNIX Command Structure

The UNIX commands follow the following format:

Command [options] [arguments]

The options/arguments are specified within square brackets if they are optional. The options are normally specified by a "-" (hyphen) followed by letter.

9.2.9 Common UNIX Commands

- Some commonly used UNIX commands are discussed below:

cal Command

The cal command creates a calendar of the specified month for the specified year, if you do not specify the month, it creates a calendar for the entire year. By default this command shows the calendar for the current month based on the system date. The cal writes its output to the standard output.

Syntax: cal [[mm] yy]

Where mm is the month, an integer between 1 and 12 and yy is the year; an integer between 1 and 9999.S

date Command

It shows or sets the system date and time. If no argument is specified, it displays the current date and the current time.

Syntax: date [+options] Options:

%d displays date as mm/dd/yy

%a displays abbreviated weekday (Sun to Sat)

%t displays time as HH:MM:SS

%r displays time as HH:MM:SS(A.M/P.M.)

%d displays only dd

%m displays only mm

If you are working in the superuser mode, you can set the date as shown below:

$ date MMddhhmm[yy]

where MM = Month (1-12), dd = day (1-31), hh = hour (1-23), mm = minutes (1-59), yy = Year

who Command

The who command lists the users that are currently logged into the system.

Syntax: who [options]

Options:

- u - lists the currently logged-in users.
- t - gives the status of all logged-in users, am
- i - this lists login-id and terminal of the user invoking this command.

finger Command

The finger command with an argument gives you more information about the user. The finger command followed by an argument can give a complete information for a user who is not logged onto the system.

Syntax: finger [user-name] Options: none

Examples

(i) $ finger xyz

If you want to know about everyone currently logged onto the system, give the following command:

$finger

The Is Command

The ls command is used for listing information about files and directories.

Syntax: ls [-options] [filename]

Options:

- -a List all directory entries including dot (.) entries.
- -d Give the name of directories only.
- -g Print group id only in the long listing.
- -i It Print inode number of each file in the first column.
- -l Lists in the long or detailed format owner's id only in the long listing.
- -s This lists the disk blocks (of 512 bytes each), occupied by a file. Sort file names by time since last access.
- -t Sort file names by time since last modified.
- -r Recursively lists all subdirectories.
- -f Marks type of each file.

The cp Command

The cp command creates a duplicate copy of a file.

Syntax: cp file1 file2

Here, the file1 is copied as file2. If file2 already exists, the new file overwrites it. **The mv Command**

This command moves or renames files.

Syntax: mv file1 file2

Here file1 refers to the source filename and 'file2' to the destination filename. Moving a file to another within the same directory is equivalent to renaming the file. mv doesn't really move the file, it just renames it and changes directory entries.

The ln Command

The 'ln' command adds one or more links to a file. Syntax: ln file1 file2

The ln command here establishes a link to an existing file. 'file1' specifies the file that has to be linked and 'file2' specifies the directory into which the link has to be established. If the 'file2' is in the same directory as file1 then the file seems to carry names, but physically there is only one copy. If you use the ls -li command, you will find that the link count has been incremented by one and that both the files have the same inode number, as they refer to the same data blocks in the disk. Any changes

that are made to one file will be reflected in the other. And if 'file2' specifies a different directory, the file will be physically present at one place but will appear as if it is present in other directory, thereby allowing different users to access the file. It saves a lot of disk space because the file is not duplicated. But you should note that you should have write permission to the directory under which the link is to be created.

The rm Command

The 'rm' command removes files or directories. Syntax: rm [options] file(s)

When you remove a file, you are actually removing a link. The space occupied by the file on the disk is freed only when you remove the last link to a file. The Options are:

c -confirms on each file before deleting it.

f - removes only those files which do not have write permission.

r - deletes the directory & its contents along with sub-directories & their contents.

The cat Command

The cat writes the contents of one of more files, onto the screen in the sequence specified. If you do not specify an input file, cat reads its data from the standard input file, generally the keyboard.

Syntax: cat file.

Examples: $ cat /usr/mkt/new-mkt.c

This command will display the contents of new-mkt.c file onto the screen.

chmod command

Using this command only a super user can change permissions (read (r), write (w), execute (x)) of user, group or others for any file on the system using the operations add (+), remove (-), and assign (=).

Examples - First see the file permissions using the ls -l command for mkt.c:

$ls -l mkt.c

Output: -rwx --x --x 2 root other 1428 May 1507:34 mkt.c

Its meaning is user has rwx, group has x and others also have x permission.

Now use the chmod command as illustrated below:

$chmod u-x g+w o+r mkt.c

The above command remove execute permission for user, give write permission to group and give read permission to all others.

Alternatively, you could have also used the following commands to do the same:

$chmod u=rw g=wx o=rx mkt.c

To assigns read, write and execute permission to all users we can use a=rwx:

$ chmod a=rwx mkt.c "

The chown Command

The chown command changes the owner of the specified file(s).

Syntax: chown new-owner filename.

This command requires you to be in the super user mode. The new owner can be the user ID of the new owner or the new owner's user number. You can also specify the owner by his name. But the new owner should have an entry in the /etc/passwd file. The filename is the name(s) of the file(s), whose owner is to be changed.

Examples: $chown bobby sales.c

The above command now makes bobby the owner of sales.c file.

The chgrp Command

The chgrp is used to change the group of a file.

Syntax: chgrp group filename.

Only the superuser can use this command. This command changes the group ownership of a file. Here group denotes the new group-ID and filename denotes the file whose group-ID is desired to be changed.

9.2.10 File System, Permissions Changing Order and Group

File is a unit of storing information. All utilities, applications and data are represented as files. The file may contain executable programs, texts or databases.

Naming Files

Filenames can be up to 14 characters long and may contain alphabets, digits and a few special characters. Files in UNIX do not have the concept of primary or secondary name as in DOS, and therefore file names may contain more than one period(.). However, UNIX file names are case sensitive.

Types

The files under UNIX can be categorized as follows:

1. Ordinary files.
2. Directory files.
3. Special files.
4. FIFO files.

Ordinary Files: They may contain executable programs, text or databases. You can add, modify or delete them or remove the file entirely.

Directory Files: Directory files represent a group of files. They contain list of file names and other information related to these files.

Special Files: Special files are also referred to as device files. These files represent physical devices such as terminals, disks, printers and tape-drives etc. These files are read from or written into just like ordinary files, except that operation on these files activates some physical devices. These files can be of two types Character device files and block device files. In character device files data is handled character by character, as in case of terminals and printers. In block device files, data is handled in large chunks of blocks, as in the case of disks and tapes.

FIFO (first-in-first-out) Files: FIFO are files that allow unrelated processes to communicate with each other. They are generally used in applications where the communication path is in only one direction, and several processes need to communicate with a single process. For example pipe in UNIX which allows transfer of data between processes in a FIFO manner. A pipe takes the output of the first process as the input to the next process, and so on.

File Names and Meta characters: In UNIX, we can refer to a group of files with the help of METACHARACTERS. The valid meta characters are ?, [, and]. It replaces any number of characters including a null character.

? - used in place of one and only one character.

[] - brackets are used to specify a set of a range of characters.

Examples

(i) $ls []c

It will list all files starting with any character or characters and ending with the character c.

(ii) $ ls robin[]

It will list all the files starting with robin and ending with any character.

(iii) $ ls x?yz[]

It will list all those files, in which the first character is x, the second character can be anything; the third and fourth characters should be respectively y and z and that the rest of the name can be anything.

(iv) $ls l[abc]mn

It will list all those files, in which the first character is l, the second character can be a, b or c, the last two characters should be m and n respectively. Alternatively the above command can also be given in the following manner.

$ ls l[a-c]mn

File Security and Ownership

The data is centralized on a system working with UNIX. The first step towards data security is the usage of passwords. The next step should be to guard the data among users. If the number of users is small, it is not much of a problem. But it can be problematic on a large system supporting many users.

UNIX can thus differentiate files belonging to an individual, the owner of a file or group of users or the others with different limited accesses, as the case may be. The different ways to access a file are:

Read(r) - You can just look through the file.

Write(w) - You can also modify it.

Execute(x) - You can just execute it

Therefore if you have a file called vendor.c and that you are the owner of it, you may provide; yourself with all the rights rwx [read, write and execute]. You can provide rx (read, and execute) rights to the members of your group and only the x (execute) right to all others.

Normally, when you create a file, you are the owner of the file and your group becomes the group id for the file. The user can also change these permissions at his will. But only a super user can change these permissions (rwx), ownership and group id's of any file in the system.

9.2.11 UNIX Editors

UNIX text editors can be classified into two types -Line Editors & Screen Editors.

(a) Line Editors: The early UNIX editors that edit processes one line at a time are called Line Editors. The common examples of UNIX line editors are ed and ex.

(i) ed - ed was the first line editor of UNIX. ed was popular in those days when most UNIX commands consisted of only two or three letters. It has become outdated now due to the use of screen editors that provides much more features.

(ii) ex - ex is more powerful and comprehensive than ed line editor. Some ex line-oriented commands are also used in few screen editors (such as vi).

(b) Screen Editors: Editors that make use of the whole screen for editing or processing more than one line at a time, are called screen editors. With screen editors, you can display and edit many lines by giving a single command. The common examples of screen editors are vi and emacs.

- vi -vi stands for 'Visual editor'. vi is the standard full-screen UNIX tool and is the only editor available on SCO UNIX.
- emacs - Although most vendors distribute emacs with UNIX system, emacs is not a part of UNIX.

9.3 Keywords

Multi-tasking: More than one program can be made to run at the same time.

Multi-user: More than one user can work at the same computer system at the same time.

Kernel: This is the actual operating system, a single large program that always resides in memory. Sections of the code in this program are executed on behalf of users to do needed tasks. Strictly speaking, the kernel is UNIX.

UNIX shell: A shell is the user-interface to the UNIX.

9.4 Summary

Unix is a multi-user, multi tasking operating system written in high-level language. It is portable, modifiable and understandable. Some popular versions of UNIX are AIX IBM, XENIX, ULTRIX, Sun OS, and BSD. The UNIX is organized as a set of layers. In the center, Kernel surrounds the hardware, which is surrounded by shell and editors that interact with the Kernel by invoking a well-defined set of system calls. Another important feature of UNIX is its security implemented by password, file permissions, and encryption. Files are organized in tree-structured directories.

Directories are also files that contain information on how to find other files. Kernel does not impose any structure on files; the meaning of bytes depends solely on the program that interprets the file. This is not true of just disc files but of peripherals devices as well, each of these is just a sequence of bytes.

UNIX provides a number of line and screen editors such as ed, ex, and vi.

9.5 SELF-ASSESSMENT QUESTIONS (SAQ)

1. How many types of files there can be in UNIX?
2. How can security of files be maintained?
3. What are different types of users in UNIX for files?
4. What is function of ls command? Give various options.
5. Differentiate between cp & mv Command.
6. What is rm command used for?
7. Differentiate between chmod & chown Command.
8. What is meant by a multi-user and multi-tasking operating system? Is UNIX multi-user, how?
9. What are various Features of UNIX? What makes UNIX portable and secure?
10. Give different layers of UNIX Architecture. Explain the intended purposes of each.
11. What do you understand by Kernel? What are the functions provided by it?
12. How can the exceptions be resolved in UNIX?
13. How are file organized in UNIX? What is the difference between a directory & a file in Unix?
14. What do you understand by Editor? Why it is needed? How many types of Editors are present?

9.6 SUGGESTED READINGS / REFERENCE MATERIAL

1. The Design of the UNIX Operating System, Bach M.J., PHI, New Delhi, 2000.
2. Operating System Concepts, 5th Edition, Silberschatz A., Galvin P.B., John Wiley & Sons.
3. Systems Programming & Operating Systems, 2nd Revised Edition, Dhamdhere D.M., Tata McGraw Hill Publishing Company Ltd., New Delhi.

4. Operating Systems, Madnick S.E., Donovan J.T., Tata McGraw Hill Publishing Company Ltd., New Delhi.
5. Operating Systems-A Modern Perspective, Gary Nutt, Pearson Education Asia, 2000.
6. Operating Systems, Harris J.A., Tata McGraw Hill Publishing Company Ltd., New Delhi, 2002.