Lesson Number: 1	***Writer: Dr. Rakesh Kumar***
Introduction to Operating System	***Vetter: Prof. Dharminder Kr.***

1.0 OBJECTIVE

The objective of this lesson is to make the students familiar with the basics of operating system. After studying this lesson they will be familiar with:

1. What is an operating system?
2. Important functions performed by an operating system.
3. Different types of operating systems.

1. 1 INTRODUCTION

Operating System (OS) is system software, which acts as an interface between a user of the computer and the computer hardware. The main purpose of an Operating System is to provide an environment in which we can execute 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.

Operating System may be viewed as collection of software consisting of procedures for operating the computer and providing an environment for execution of programs. It is an interface between user and computer. So an Operating System makes everything in the computer to work together smoothly and efficiently.

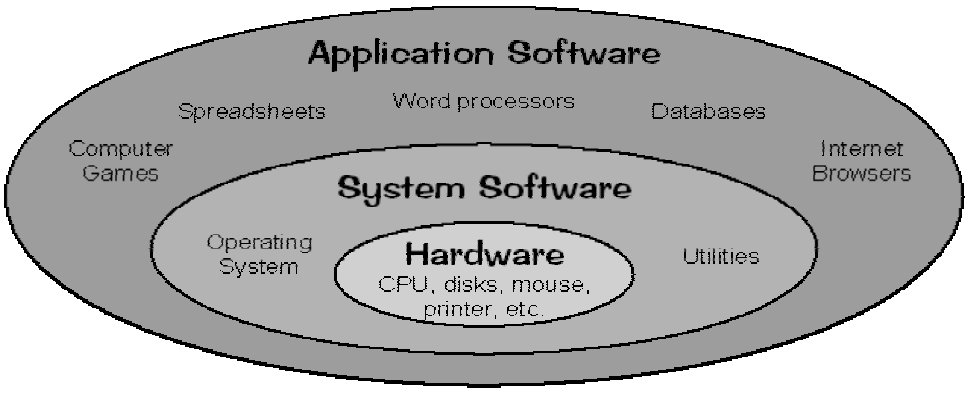


Figure 1: The relationship between application and system software

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.

The Operating System is also responsible for security and ensuring that unauthorized users do not access the system. Figure 1 illustrates the relationship between application software and system software.

The first two responsibilities address the need for managing the computer hardware and the application programs that use the hardware. The third responsibility focuses on providing an interface between application software and hardware so that application software can be efficiently developed. Since the Operating System is already responsible for managing the hardware, it should provide a programming interface for application developers. As a user, we normally interact with the Operating System through a set of commands. The commands are accepted and executed by a part of the Operating System called the command processor or command line interpreter.

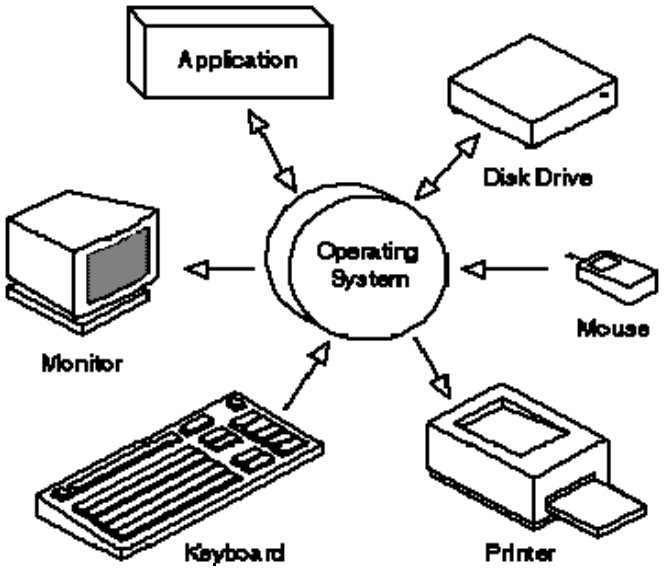


Figure 2: The interface of various devices to an operating system

In order to understand operating systems we must understand the computer hardware and the development of Operating System from beginning. Hardware means the physical machine and its electronic components including memory chips, input/output devices, storage devices and the central processing unit. Software are the programs written for these computer systems. Main memory is where the data and instructions are stored to be processed. Input/Output devices are the peripherals attached to the system, such as keyboard, printers, disk drives, CD drives, magnetic tape drives, modem, monitor, etc. The central processing unit is the brain of the computer system; it has circuitry to control the interpretation and execution of instructions. It controls the operation of entire computer system. All of the storage references, data manipulations and I/O operations are performed by the CPU. The entire computer systems can be divided into four parts or components (1) The hardware (2) The Operating System (3) The application programs and system programs (4) The users.

The hardware provides the basic computing power. The system programs the way in which these resources are used to solve the computing problems of the users. There may be many different users trying to solve different problems. The Operating System controls and coordinates the use of the hardware among the various users and the application programs.

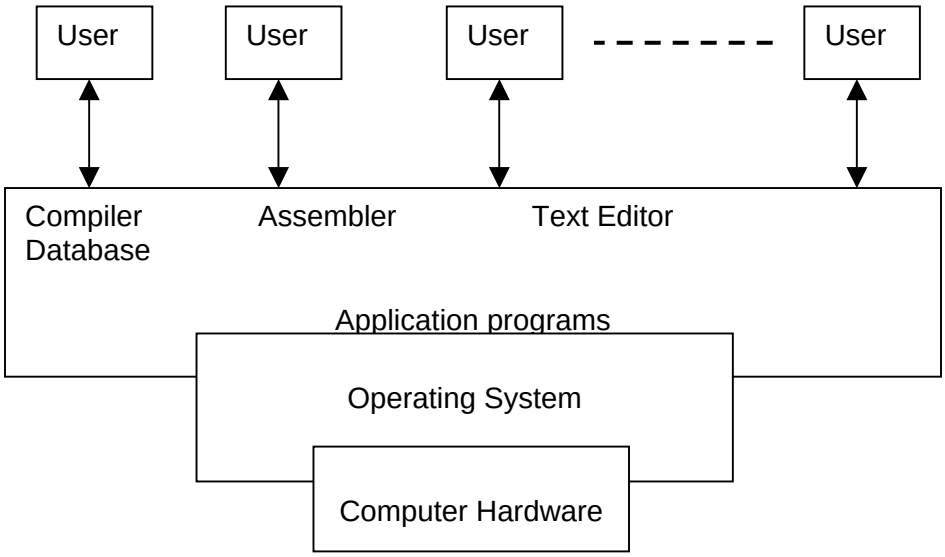
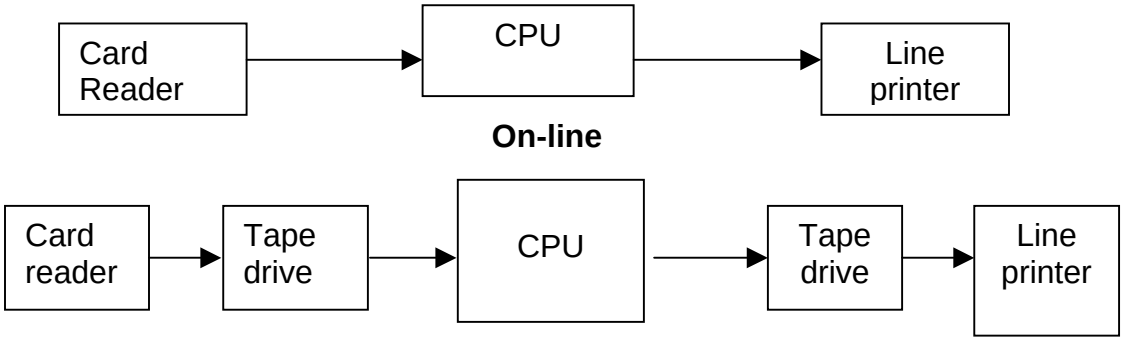


Figure 3. Basic components of a computer system

We can view an Operating System as a resource allocator. A computer system has many resources, which are to be required to solve a computing problem. These resources are the CPU time, memory space, files storage space, input/output devices and so on. The Operating System acts as a manager of all of these resources and 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.

An Operating System can also be viewed as a control program, used to control the various I/O devices and the users programs. A control program controls the execution of the user programs to prevent errors and improper use of the computer resources. It is especially concerned with the operation and control of I/O devices. As stated above the fundamental goal of computer system is to execute user programs and solve user problems. For this goal computer hardware is constructed. But the bare hardware is not easy to use and for this purpose application/system programs are developed. These various programs require some common operations, such as controlling/use of some input/output devices and the use of CPU time for execution. The common functions of controlling and allocation of resources between different users and application programs is brought together into one piece of software called operating system. It is easy to define operating systems by what they do rather than what they are. The primary goal of the operating systems is convenience for the user to use the computer. Operating systems makes it easier to compute. A secondary goal is efficient operation of the computer system. The large computer systems are very expensive, and so it is desirable to make them as efficient as possible. Operating systems thus makes the optimal use of computer resources. In order to understand what operating systems are and what they do, we have to study how they are developed. Operating systems and the computer architecture have a great influence on each other. To facilitate the use of the hardware operating systems were developed.

First, professional computer operators were used to operate the computer. The programmers no longer operated the machine. As soon as one job was finished, an operator could start the next one and if some errors came in the program, the operator takes a dump of memory and registers, and from this the programmer have to debug their programs. The second major solution to reduce the setup time was to batch together jobs of similar needs and run through the computer as a group. But there were still problems. For example, when a job stopped, the operator would have to notice it by observing the console, determining why the program stopped, takes a dump if necessary and start with the next job. To overcome this idle time, automatic job sequencing was introduced. But even with batching technique, the faster computers allowed expensive time lags between the CPU and the I/O devices. Eventually several factors helped improve the performance of CPU. First, the speed of I/O devices became faster. Second, to use more of the available storage area in these devices, records were blocked before they were retrieved. Third, to reduce the gap in speed between the I/O devices and the CPU, an interface called the control unit was placed between them to perform the function of buffering. A buffer is an interim storage area that works like this: as the slow input device reads a record, the control unit places each character of the record into the buffer. When the buffer is full, the entire record is transmitted to the CPU. The process is just opposite to the output devices. Fourth, in addition to buffering, an early form of spooling was developed by moving off-line the operations of card reading, printing etc. SPOOL is an acronym that stands for the simultaneous peripherals operations on-line. For example, incoming jobs would be transferred from the card decks to tape/disks off-line. Then they would be read into the CPU from the tape/disks at a speed much faster than the card reader.



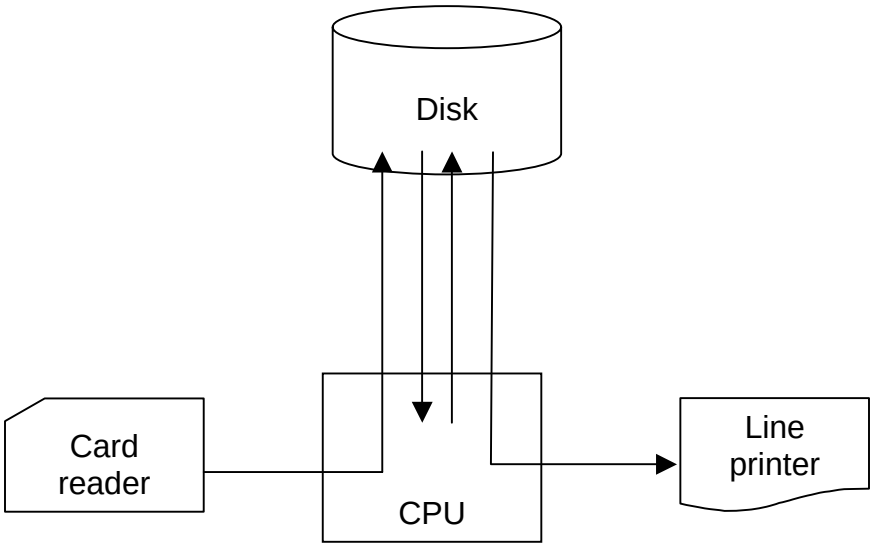


Figure 4: the on-line, off-line and spooling processes

Moreover, the range and extent of services provided by an Operating System depends on a number of factors. Among other things, the needs and characteristics of the target environmental that the Operating System is intended to support largely determine user- visible functions of an operating system. For example, an Operating System intended for program development in an interactive environment may have a quite different set of system calls and commands than the Operating System designed for run-time support of a car engine.

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 Device Management Functions

1.2.1.4 Information Management Functions

1.2.2 Evolution of Processing Trends

1.2.2.1 Serial Processing

1.2.1 Operating System as a Resource Manager

The Operating System is a manager of system resources. A computer system has many resources as stated above. 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. *Here we present a framework of the study of Operating System based on the view that the Operating System is manager of resources. The Operating System as a resources manager can be classified in to the following three popular views: primary view, hierarchical view, and extended machine view.*

The primary view is that the Operating System is a collection of programs designed to manage the system's resources, namely, memory, processors, peripheral devices, and information. It is the function of Operating System to see that they are used efficiently and to resolve conflicts arising from competition among the various users. The Operating System must keep track of status of each resource; decide which process is to get the resource, allocate it, and eventually reclaim it.

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. In multiprogramming environment, multiple programs are maintained in the memory simultaneously. The Operating System is responsible for the following memory management functions:

- Keep track of which segment of memory is in use and by whom.
- Deciding which processes are to be loaded into memory when space becomes available. In multiprogramming environment it decides which process gets the available memory, when it gets it, where does it get it, and how much.
- 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. To accomplish its task, 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:

- Provides mechanisms for process synchronization for sharing of resources amongst concurrent processes.
- Keeps track of processor and status of processes. The program that does this has been called the traffic controller.
- Decide which process will have a chance to use 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. The module that does this is called a process scheduler.

- Allocate the processor to a process by setting up the necessary hardware registers. This module is widely known as the dispatcher.
- Providing mechanisms for deadlock handling.
- 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

An Operating System will have device drivers to facilitate I/O functions involving I/O devices. These device drivers are software routines that control respective I/O devices through their controllers. The Operating System is responsible for the following I/O Device Management Functions:

- Keep track of the I/O devices, I/O channels, etc. This module is typically called I/O traffic controller.
- 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.
- Allocate the I/O device and initiate the I/O operation.
- Reclaim device as and when its use is through. In most cases I/O terminates automatically.

1.2.1.4 Information Management Functions

- Keeps track of the information, its location, its usage, status, etc. The module called a file system provides these facilities.
- Decides who gets hold of information, enforce protection mechanism, and provides for information access mechanism, etc.
- Allocate the information to a requesting process, e.g., open a file.
- 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, increased

functionality and enhanced reliability. Various networking protocols are TCP/IP (Transmission Control Protocol/ Internet Protocol), UDP (User Datagram Protocol), FTP (File Transfer Protocol), HTTP (Hyper Text Transfer protocol), NFS (Network File System) etc.

1.2.2 EVOLUTION OF PROCESSING TRENDS

Starting from the bare machine approach to its present forms, the Operating System has evolved through a number of stages of its development like serial processing, batch processing multiprocessing etc. as mentioned below:

1.2.2.1 Serial Processing

In theory, every computer system may be programmed in its machine language, with no systems software support. Programming of the bare machine was customary for early computer systems. A slightly more advanced version of this mode of operation is common for the simple evaluation boards that are sometimes used in introductory microprocessor design and interfacing courses. Programs for the bare machine can be developed by manually translating sequences of instructions into binary or some other code whose base is usually an integer power of 2. Instructions and data are then entered into the computer by means of console switches, or perhaps through a hexadecimal keyboard. Loading the program counter with the address of the first instruction starts programs. Results of execution are obtained by examining the contents of the relevant registers and memory locations. The executing program, if any, must control Input/output devices, directly, say, by reading and writing the related I/O ports. Evidently, programming of the bare machine results in low productivity of both users and hardware. The long and tedious process of program and data entry practically precludes execution of all but very short programs in such an environment.

The next significant evolutionary step in computer-system usage came about with the advent of input/output devices, such as punched cards and paper tape, and of language translators. Programs, now coded in a programming language, are translated into executable form by a computer program, such as a compiler or an interpreter. Another program, called the loader, automates the process of

loading executable programs into memory. The user places a program and its input data on an input device, and the loader transfers information from that input device into memory. After transferring control to the loader program by manual or automatic means, execution of the program commences. The executing program reads its input from the designated input device and may produce some output on an output device. Once in memory, the program may be rerun with a different set of input data.

The mechanics of development and preparation of programs in such environments are quite slow and cumbersome due to serial execution of programs and to numerous manual operations involved in the process. In a typical sequence, the editor program is loaded to prepare the source code of the user program. The next step is to load and execute the language translator and to provide it with the source code of the user program. When serial input devices, such as card reader, are used, multiple-pass language translators may require the source code to be repositioned for reading during each pass. If syntax errors are detected, the whole process must be repeated from the beginning. Eventually, the object code produced from the syntactically correct source code is loaded and executed. If run-time errors are detected, the state of the machine can be examined and modified by means of console switches, or with the assistance of a program called a debugger.

1.2.2.2 Batch Processing

With the invention of hard disk drive, the things were much better. The batch processing was relied on punched cards or tape for the input when assembling the cards into a deck and running the entire deck of cards through a card reader as a batch. Present batch systems are not limited to cards or tapes, but the jobs are still processed serially, without the interaction of the user. The efficiency of these systems was measured in the number of jobs completed in a given amount of time called as throughput. Today's operating systems are not limited to batch programs. This was the next logical step in the evolution of operating systems to automate the sequencing of operations involved in program execution and in the mechanical aspects of program development. The intent was to increase system

resource utilization and programmer productivity by reducing or eliminating component idle times caused by comparatively lengthy manual operations.

Furthermore, even when automated, housekeeping operations such as mounting of tapes and filling out log forms take a long time relative to processors and memory speeds. Since there is not much that can be done to reduce these operations, system performance may be increased by dividing this overhead among a number of programs. More specifically, if several programs are batched together on a single input tape for which housekeeping operations are performed only once, the overhead per program is reduced accordingly. A related concept, sometimes called phasing, is to prearrange submitted jobs so that similar ones are placed in the same batch. For example, by batching several Fortran compilation jobs together, the Fortran compiler can be loaded only once to process all of them in a row. To realize the resource-utilization potential of batch processing, a mounted batch of jobs must be executed automatically, without slow human intervention. Generally, Operating System commands are statements written in Job Control Language (JCL). These commands are embedded in the job stream, together with user programs and data. A memory-resident portion of the batch operating system- sometimes called the batch monitor- reads, interprets, and executes these commands.

Moreover, the sequencing of program execution mostly automated by batch operating systems, the speed discrepancy between fast processors and comparatively slow I/O devices, such as card readers and printers, emerged as a major performance bottleneck. Further improvements in batch processing were mostly along the lines of increasing the throughput and resource utilization by overlapping input and output operations. These developments have coincided with the introduction of direct memory access (DMA) channels, peripheral controllers, and later dedicated input/output processors. As a result, computers for offline processing were often replaced by sophisticated input/output programs executed on the same computer with the batch monitor.

Many single-user operating systems for personal computers basically provide for serial processing. User programs are commonly loaded into memory and

executed in response to user commands typed on the console. A file management system is often provided for program and data storage. A form of batch processing is made possible by means of files consisting of commands to the Operating System that are executed in sequence. Command files are primarily used to automate complicated customization and operational sequences of frequent operations.

1.2.2.3 Multiprogramming

In multiprogramming, many processes are simultaneously resident in memory, and execution switches between processes. The advantages of multiprogramming are the same as the commonsense reasons that in life you do not always wait until one thing has finished before starting the next thing. Specifically:

- More efficient use of computer time. If the computer is running a single process, and the process does a lot of I/O, then the CPU is idle most of the time. This is a gain as long as some of the jobs are I/O bound -- spend most of their time waiting for I/O.
- Faster turnaround if there are jobs of different lengths. Consideration (1) applies only if some jobs are I/O bound. Consideration (2) applies even if all jobs are CPU bound. For instance, suppose that first job A, which takes an hour, starts to run, and then immediately afterward job B, which takes 1 minute, is submitted. If the computer has to wait until it finishes A before it starts B, then user A must wait an hour; user B must wait 61 minutes; so the average waiting time is 60-1/2 minutes. If the computer can switch back and forth between A and B until B is complete, then B will complete after 2 minutes; A will complete after 61 minutes; so the average waiting time will be 31-1/2 minutes. If all jobs are CPU bound and the same length, then there is no advantage in multiprogramming; you do better to run a batch system. The multiprogramming environment is supposed to be invisible to the user processes; that is, the actions carried out by each process should proceed in the same was as if the process had the entire machine to itself.

This raises the following issues:

- Process model: The state of an inactive process has to be encoded and saved in a process table so that the process can be resumed when made active.
- Context switching: How does one carry out the change from one process to another?
- Memory translation: Each process treats the computer's memory as its own private playground. How can we give each process the illusion that it can reference addresses in memory as it wants, but not have them step on each other's toes? The trick is by distinguishing between virtual addresses -- the addresses used in the process code -- and physical addresses -- the actual addresses in memory. Each process is actually given a fraction of physical memory. The memory management unit translates the virtual address in the code to a physical address within the user's space. This translation is invisible to the process.
- Memory management: How does the Operating System assign sections of physical memory to each process?
- Scheduling: How does the Operating System choose which process to run when?

Let us briefly review some aspects of program behavior in order to motivate the basic idea of multiprogramming. This is illustrated in Figure 6, indicated by dashed boxes. Idealized serial execution of two programs, with no inter-program idle times, is depicted in Figure 6(a). For comparison purposes, both programs are assumed to have identical behavior with regard to processor and I/O times and their relative distributions. As Figure 6(a) suggests, serial execution of programs causes either the processor or the I/O devices to be idle at some time even if the input job stream is never empty. One way to attack this problem is to assign some other work to the processor and I/O devices when they would otherwise be idling.

Program 1 Program 2

Figure 6(a)

Figure 6(b) illustrates a possible scenario of concurrent execution of the two programs introduced in Figure 6(a). It starts with the processor executing the first computational sequence of Program 1. Instead of idling during the subsequent I/O sequence of Program 1, the processor is assigned to the first computational sequence of the Program 2, which is assumed to be in memory and awaiting execution. When this work is done, the processor is assigned to Program 1 again, then to Program 2, and so forth.

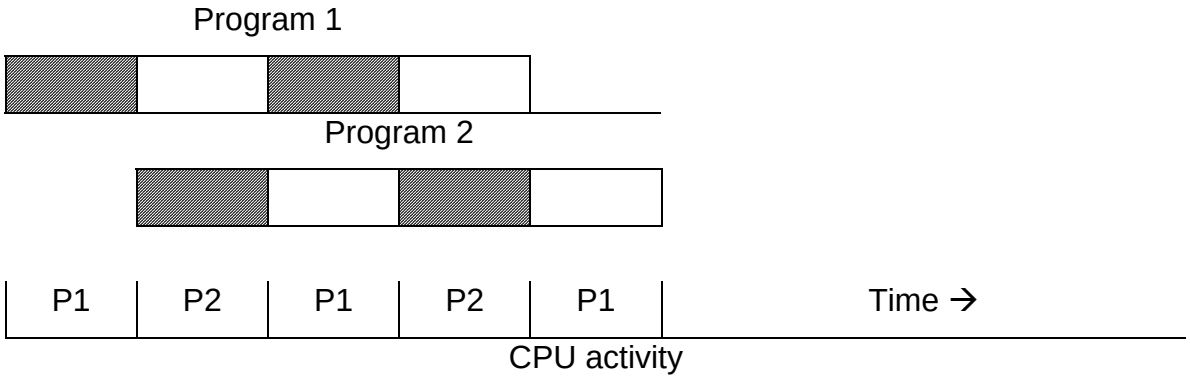


Figure 6 (b) Multiprogrammed executions

As Figure 6 suggests, significant performance gains may be achieved by interleaved executing of programs, or multiprogramming, as this mode of operation is usually called. With a single processor, parallel execution of programs is not possible, and at most one program can be in control of the processor at any time. The example presented in Figure 6(b) achieves 100% processor utilization with only two active programs. The number of programs actively competing for resources of a multi-programmed computer system is called the degree of multiprogramming. In principle, higher degrees of multiprogramming should result in higher resource utilization. Time-sharing systems found in many university computer centers provide a typical example of a multiprogramming system.

1.2.3 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. In the following discussion several types of operating systems are discussed**.**

1.2.3.1 Batch Operating System

Batch processing is the most primitive type of operating system. Batch processing generally requires the program, data, and appropriate system commands to be submitted together in the form of a job. Batch operating systems usually allow little or no 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 programs with long execution times may be served well by a batch operating system. Examples of such programs include payroll, forecasting, statistical analysis, and large scientific number-crunching programs. Serial processing combined with batch like command files is also found on many personal computers. Scheduling in batch is very simple. Jobs are typically processed in order of their submission, that is, first-come first-served fashion.

Memory management in batch systems is also very simple. 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. For this reason, many serial and I/O and ordinary batch operating systems use simple, program controlled method of I/O. The lack of contention for I/O devices makes their allocation and deallocation trivial.

Batch systems often provide simple forms of file management. Since access to files is also serial, little protection and no concurrency control of file access in required.

1.2.3.2 Multiprogramming Operating System

A multiprogramming system permits multiple programs to be loaded into memory and execute the programs concurrently. Concurrent execution of programs has a significant potential for improving system throughput and resource utilization

relative to batch and serial processing. 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.

1.2.3.3 Multitasking Operating System

It allows more than one program to run concurrently. The ability to execute more than one task at the same time is called as multitasking. An instance of a program in execution is called a process or a task. A multitasking Operating System is distinguished by its ability to support concurrent execution of two or more active processes. 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. OS/2, Windows 95, Windows NT, and UNIX use preemptive multitasking, whereas Microsoft Windows 3.x and the MultiFinder use cooperative multitasking.

1.2.3.4 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.

1.2.3.5 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.

1.2.3.6 Time-sharing system

Time-sharing is a popular representative of multi-programmed, multi-user systems. In addition to general program-development environments, many large computer-aided design and text-processing systems belong to this category. One of the primary objectives of multi-user systems in general, and time-sharing in particular, is good terminal response time. Giving the illusion to each user of having a machine to oneself, time-sharing systems often attempt to provide equitable sharing of common resources. For example, when the system is

loaded, users with more demanding processing requirements are made to wait longer.

This philosophy is reflected in the choice of scheduling algorithm. Most time-sharing systems use time-slicing scheduling. In this approach, 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. This mode of operation generally provides quick response time to interactive 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 possibly 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. As in most multi-user environments, allocation and de-allocation of devices must be done in a manner that preserves system integrity and provides for good performance.

1.2.3.7 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. Real time systems are used for space flights, airport traffic control, industrial processes, sophisticated medical equipments, telephone switching etc. A real time system must be 100 percent responsive in time. Response time 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. If the timing constraints of the system are not met, system failure is said to have occurred. 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 in a short time or within certain deadlines.

A primary objective of real-time systems is to provide quick event-response times, and thus meet the scheduling deadlines. User convenience and resource utilization are of secondary concern to real-time system designers. 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. Such requirements usually cannot be met by multi-programming alone, and real-time operating systems usually rely on some specific policies and techniques for doing their job.
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 among those that are ready to execute. Higher-priority processes usually preempt execution of the lower-priority processes. This form of scheduling, called priority-based preemptive scheduling, is used by a majority of real-time systems. Unlike, say, time-sharing, the process population in real-time systems is fairly static, and there is comparatively little moving of programs between primary and secondary storage. On the other hand, processes in real-time systems tend to cooperate closely, thus necessitating support for both separation and sharing of memory. Moreover, as already suggested, 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 usually found only in larger installations of real-time systems. In fact, some embedded real-time systems, such as an onboard automotive controller, may not even have any secondary storage. The primary objective of file management in real-time systems is usually speed of access, rather then efficient utilization of secondary storage.

1.2.3.8 Combination of operating systems

Different types of Operating System are optimized or geared up 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.

1.2.3.9 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. Historically, 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. The key objective of a distributed Operating System is transparency. Ideally, component and resource distribution should be hidden from users and application programs unless they explicitly demand otherwise. 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 provided at each node for the benefit of local clients, a distributed Operating System may facilitate access to remote resources, communication with remote processes, and distribution of computations. The added services necessary for pooling of shared system resources include global naming, distributed file system, and facilities for distribution**.**

1.3 SUMMARY

Operating System is also known as resource manager because its prime responsibility is to manage the resources of the computer system i.e. memory, processor, devices and files. In addition to these, Operating System provides an interface between the user and the bare machine. Following the course of the conceptual evolution of operating systems, we have identified the main characteristics of the program-execution and development environments provided by the bare machine, serial processing, including batch and multiprogramming.

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.

1.4 Keywords

(i) SPOOL: Simultaneous Peripheral Operations On Line

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

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

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

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

1.5. SELF ASSESMENT QUESTIONS (SAQ)

1. What are the objectives of an operating system? Discuss.
2. Differentiate between multiprogramming, multitasking, and multiprocessing.
3. Discuss modular approach of development of an operating system.

4. Discuss whether there are any advantages of using a multitasking operating system, as opposed to a serial processing one.
5. What are the major functions performed by an operating system? Explain.
6. Why operating system is referred to as resource manager? Explain.
7. Write a detailed note on the evolution of operating systems.
8. What is a real time system? How is it different from other types of operating systems? Explain.

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***

System calls and system programs ***Vetter: Prof. Dharminder Kumar***

2.0 Objectives

The objective of this lesson is to provide the information about the various services provided by the operating system. After studying this lesson the students will be familiar with the various system services and how are those implemented.

2.1 Introduction

Operating system provides an environment in which programs are executed. Since operating system can only directly interact with the bare machine, it can only perform the basic input and output operations, so all the users programs have to request the operating system to perform these operations.

As discussed in previous lesson, there arises a need to identify the system resources that must be managed by the Operating System and using the process viewpoint, we indicate when the corresponding resource manager comes into play. We now answer the question, “How are these resource managers activated, and where do they reside?” Does memory manager ever invoke the process scheduler? Does scheduler ever call upon the services of memory manager? Is the process concept only for the user or is it used by Operating System also?

The Operating System provides many instructions in addition to the Bare machine instructions (A Bare machine is a machine without its software clothing, and it does not provide the environment which most programmers are desired for). Instructions that form a part of Bare machine plus those provided by the Operating System constitute the instruction set of the extended machine. The situation is pictorially represented in figure 1. The Operating System kernel runs on the bare machine; user programs run on the extended machine. This means that the kernel of Operating System is written by using the instructions of bare

machine only; whereas the users can write their programs by making use of instructions provided by the extended machine.

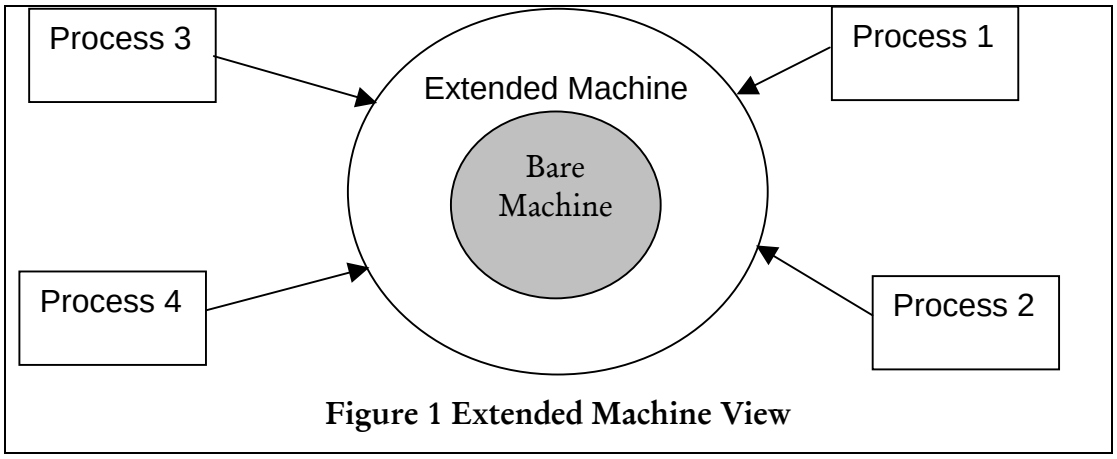


Figure 1 Extended Machine View

The Operating System kernel runs on the bare machine; user programs run on the extended machine. This means that the kernel of Operating System is written by using the instructions of bare machine only; whereas the users can write their programs by making use of instructions provided by the extended machine.

2.2 Presentation of contents

2.2.1 Hierarchical structure of an operating system

2.2.2 Virtual Machine

2.2.3 System Services

2.2.3.1 System Calls

2.2.3.1.1 Types of System Calls

2.2.3.1.2 System Call implementations

2.2.3.1.3 Common system calls

2.2.3.2 System Programs

2.2.1 Hierarchical Structure of an Operating System

Let us now discuss how the operating system is put together. Most of the early operating systems consisted simply of one big program. This was called a brute force or monolithic approach. As computers systems became larger and more comprehensive, abovementioned approach became unmanageable. A better approach is to develop an operating system employing a modular approach. In this section we discuss a hierarchical view of an operating system to show how

various modules of an Operating System are organized with respect to each other.

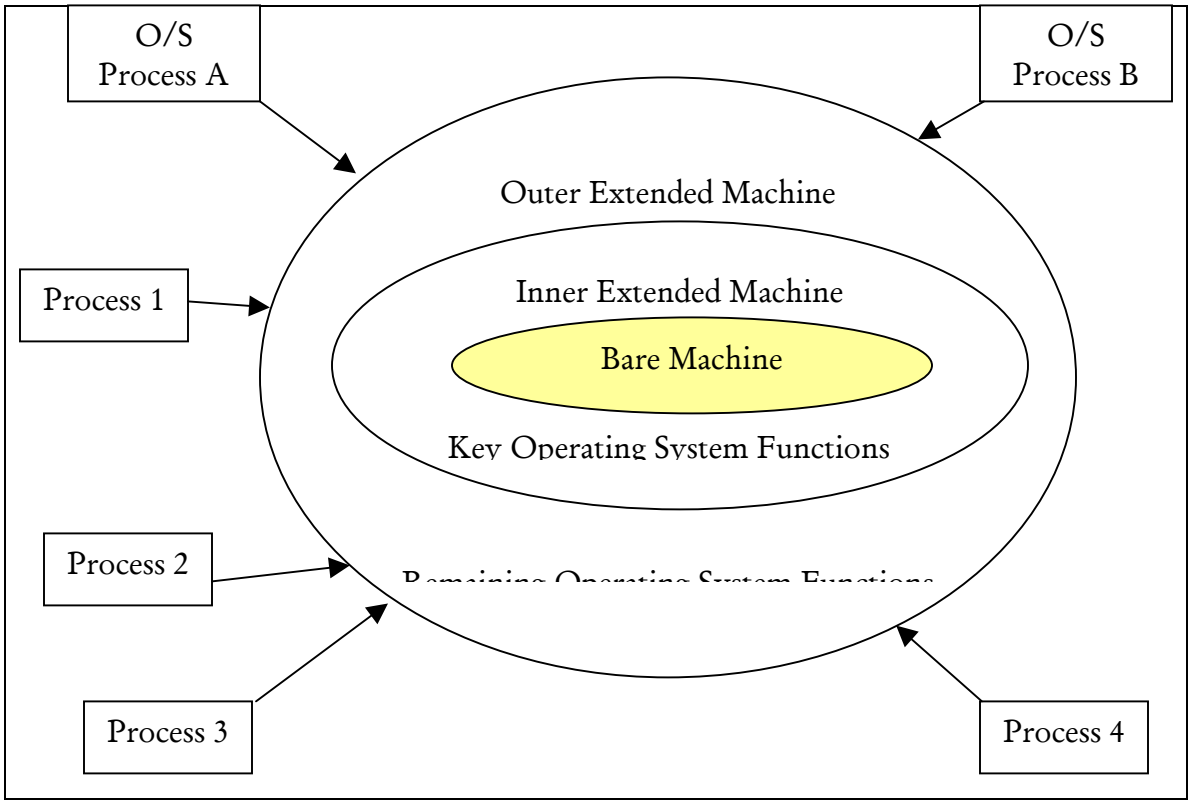


Figure 2: Simple Hierarchical Machine View

In order to use the hierarchical approach, we must answer the original question: Where does each module of the operating system fit in the hierarchy? Does it fit in the inner extended machine, or the outer extended machine, or as a process? Furthermore, the concept of two-level extended (inner and outer) machine can be extended even more; resulting into a multi-layer and multilevel approach. Figure 3 illustrates the extended hierarchical structure of an operating system. All the processes (shown in boxes) use the kernel and share all the resources of the system. The parent-child or controller-controlled relationship between processes is depicted in figure 3 by placing them in different layers.

In a strictly hierarchical implementation, a given level is allowed to call upon services of lower level, but not upon those of higher levels. In figure 3, layer0 (kernel) is divided into 5 levels.

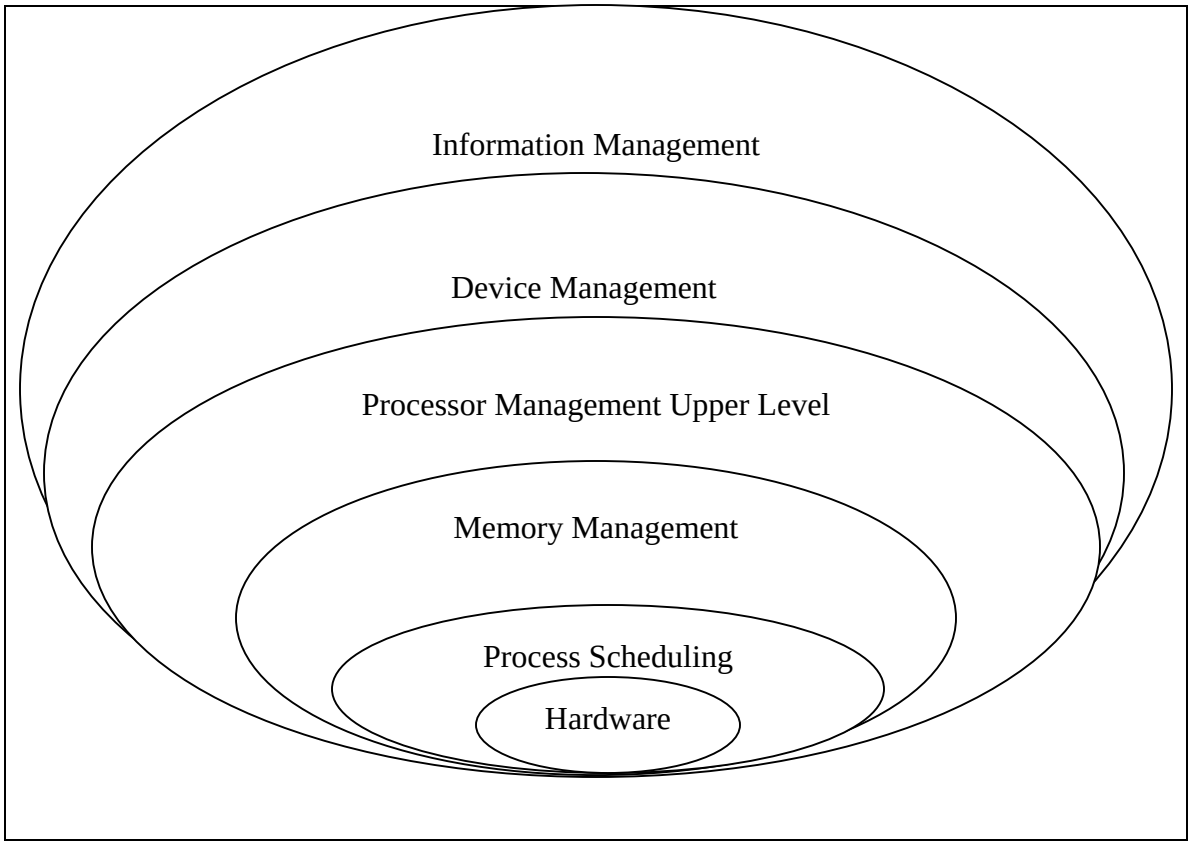


Figure 3: Hierarchical Operating System Structure

Primitive functions residing in each level is discussed below:

Level 1: Processor Management Lower Level

- P and V operators
- Process scheduling

Level 2: Memory Management

- Allocate memory
- Release memory

Level 3: Processor Management Upper Level

- Create/destroy process
- Send/receive messages between processes
- Start/stop process

Level 4: Device Management

- Keep track of status of all I/O devices
- Schedule I/O operations
- Initiate I/O process

Level 5: Information Management

Create/destroy file

Open/Close file

Read/write file

2.2.2. Virtual Machines

The virtual machine approach makes it possible to run different operating system on the same real machine.

System virtual machines (sometimes called hardware virtual machines) allow the sharing of the underlying physical machine resources between different virtual machines, each running its own operating system. The software layer providing the virtualization is called a virtual machine monitor or hypervisor. A hypervisor can run on bare hardware or on top of an operating system.

The main advantages of system Virtual Machines are:

- multiple Operating System environments can co-exist on the same computer, in strong isolation from each other
- the virtual machine can provide an instruction set architecture (ISA) that is somewhat different from that of the real machine
- Application provisioning, maintenance, high availability and disaster recovery.

Multiple Virtual Machines each running their own operating system (called guest operating system) are frequently used in server consolidation, where different services that used to run on individual machines in order to avoid interference are instead run in separate Virtual Machines on the same physical machine. This use is frequently called quality-of-service isolation (QoS isolation).

The desire to run multiple operating systems was the original motivation for virtual machines, as it allowed time-sharing a single computer between several single-tasking Operating Systems. This technique requires a process to share the CPU resources between guest operating systems and memory virtualization to share the memory on the host.

The guest Operating Systems do not have to be all the same, making it possible to run different Operating Systems on the same computer (e.g., Microsoft

Windows and Linux, or older versions of an Operating System in order to support software that has not yet been ported to the latest version). The use of virtual machines to support different guest Operating Systems is becoming popular in embedded systems; a typical use is to support a real-time operating system at the same time as a high-level Operating System such as Linux or Windows.

Another use is to sandbox an Operating System that is not trusted, possibly because it is a system under development. Virtual machines have other advantages for Operating System development, including better debugging access and faster reboots.

Consider the following figure in which OS1, OS2, and OS4 are three different operating systems and OS3 is operating system under test. All these operating systems are running on the same real machine but they are not directly dealing with the real machine, they are dealing with Virtual Machine Monitor (VMM) which provides each user with the illusion of running on a separate machine. If the operating system being tested causes a system to crash, this crash affects only its own virtual machine. The other users of the real machine can continue their operation without being disturbed. Actually lowest level routines of the operating system deals with the VMM instead of the real machine which provides the services and functions as those available on the real machine. Each user of the virtual machine i.e. OS1, OS2 etc. runs in user mode, not supervisor mode, on the real machine.

User 1	User 2	Test User	User 4
Operating System OS1	Operating System OS2	Operating System OS3 (test)	Operating System OS4
Virtual Machine Monitor (VMM)			
Real Machine			

Figure 4: Multiple users of a virtual machine operating system

2.2.3 System Services

An operating system provides an environment for the execution of the programs. It provides certain services to programs and the users of the programs. The services are:

(a) Program Execution: If a user want to execute a program then system must be able to load it in memory and run it. The program must be able to end it execution.

(b) I/O operations: The running program may require input and output such as a file or an I/O device. The program cannot execute I/O operation directly, so the OS must facilitate this thing.

(c) File system manipulation: If the running program is in need of files the OS should facilitate creation, deletion etc of files.

(d) Error detection: The operating system has to continuously monitor the system because error may occur at any place such as in the CPU, in memory, in I/O devices or in the user program itself. The operating system has to ensure the correct and continuous computing.

In addition to above classes of services, a number of other services are resource allocation, accounting and protection. When there are multiple users and programs the operating system has to manage the resources and keep account of each user and resources occupied by them. When multiple jobs are there in the system, operating system has to ensure that one should not interfere with the others.

The two most common approaches to provide the services are system calls and system programs.

2.2.3.1 SYSTEM CALLS

> *The interface between the operating system and the user programs is defined by the set of "extended instructions" that the operating system provides. These extended instructions are known as system calls.*

System calls provide an interface between the process and the operating system. System calls allow user-level processes to request some services from the operating system which process itself is not allowed to do. In handling the trap, the operating system will enter in the kernel mode, where it has access to

privileged instructions, and can perform the desired service on the behalf of user-level process. It is because of the critical nature of operations that the operating system itself does them every time they are needed. For example, for I/O a process involves a system call telling the operating system to read or write particular area and this request is satisfied by the operating system.

System programs provide basic functioning to users so that they do not need to write their own environment for program development (editors, compilers) and program execution (shells). In some sense, they are bundles of useful system calls

2.2.3.1.1 Types of System Calls

System calls are kernel level service routines for implementing basic operations performed by the operating system. System calls can be grouped into three major categories:

(a) Process and job control

(b) Device and file manipulation

(c) Information maintenance

Process and job control

The category includes the system call to end or abort the running program, to load and execute the program, to create new process or terminate the existing one, to get the process attributes and to set them. Another set of the system calls are helpful in debugging a program and to dump the memory.

File manipulation

Systems calls are required to read and delete the file, to open them and to close them. In order to perform the read, write and reposition operations we need the system calls. To read and determine the attributes of the files we need system calls.

Device management

In order to use a device, we first request the device, after using it we have to release it. Once the device has been requested we can read, write and reposition the device.

Information maintenance

Many system calls exist for the purpose of transferring information between the user program and operating system such as a call to return the current time and date.

Types of system calls	
Process Control	
1.	End, Abort
2.	Load, Execute
3.	Create Process, Terminate Process
4.	Get and Set Process attributes
File manipulation	
1.	Create File, Delete file
2.	Open and close file
3.	Read, write and reposition
4.	Get and set file attributes
Device manipulation	
1.	Request and release the devices
2.	Read, write and reposition
3.	Get and set Device attributes
Information Maintenance	
1.	Get/Set time or date
2.	Get/Set system date
3.	Get/Set process/file/device attributes

A system call is a request made by any program to the operating system for performing tasks -- picked from a predefined set -- which the said program does not have required permissions to execute in its own flow of execution. System calls provide the interface between a process and the operating system. Most operations interacting with the system require permissions not available to a user level process, e.g. I/O performed with a device present on the system or any form of communication with other processes requires the use of system calls.

The fact that improper use of the system call can easily cause a system crash necessitates some level of control. The design of the microprocessor architecture

on practically all modern systems (except some embedded systems) offers a series of *privilege levels* -- the (low) privilege level in which normal applications execute limits the address space of the program so that it cannot access or modify other running applications nor the operating system itself. It also prevents the application from directly using devices (e.g. the frame buffer or network devices). But obviously many normal applications need these abilities; thus they can call the operating system. The operating system executes at the highest level of privilege and allows the applications to request services via system calls, which are often implemented through interrupts. If allowed, the system enters a higher privilege level, executes a specific set of instructions which the interrupting program has no direct control over, then returns control to the former flow of execution. This concept also serves as a way to implement security.

With the development of separate operating modes with varying levels of privilege, a mechanism was needed for transferring control safely from lesser privileged modes to higher privileged modes. Less privileged code could not simply transfer control to more privileged code at any point and with any processor state. To allow it to do so would allow it to break security. For instance, the less privileged code could cause the higher privileged code to execute in the wrong order, or provide it with a bad stack.

The library as an intermediary

Generally, systems provide a library that sits between normal programs and the operating system, usually an implementation of the C library (libc), such as glibc. This library handles the low-level details of passing information to the operating system and switching to supervisor mode, as well as any data processing and preparation which does not need to be done in privileged mode. Ideally, this reduces the coupling between the Operating System and the application, and increases portability.

2.2.3.1.2 System Call implementations

On Unix, Unix-like and other POSIX-compatible Operating Systems, popular system calls are open, read, write, close, wait, exec, fork, exit, and kill. Many of

today's operating systems have hundreds of system calls. For example, Linux has 319 different system calls.

Implementing system calls requires a control transfer which involves some sort of architecture-specific feature. A typical way to implement this is to use a software interrupt or trap. Interrupts transfer control to the Operating System so software simply needs to set up some register with the system call number they want and execute the software interrupt.

Often more information is required then simply the call number. The exact type and amount of information depend upon the operating system and call. Three general methods are used to pass parameters between a running program and the operating system.

- Pass parameters in registers.
- Store the parameters in a table in memory, and the table address is passed as a parameter in a register.
- Push (store) the parameters onto the stack by the program, and pop off the stack by operating system.

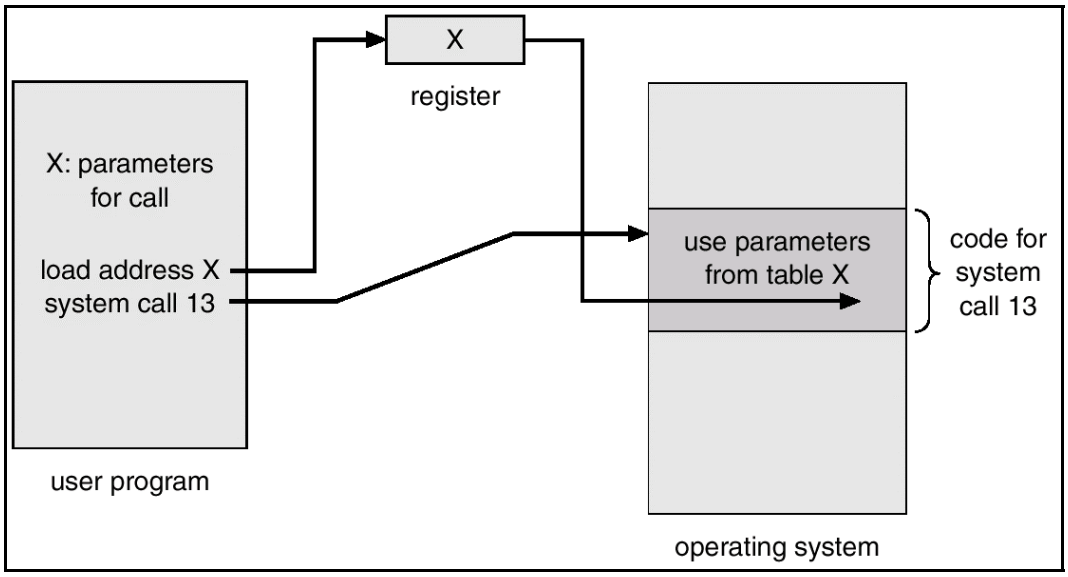


Figure 5: Passing Parameters

2.2.3.1.3 Common system calls

Below are mentioned some of several generic system calls that most operating systems provide.

CREATE (processID, attributes);

In response to the CREATE call, the Operating System creates a new process with the specified or default attributes and identifier. A process cannot create itself-because it would have to be running in order to invoke the Operating System, and it cannot run before being created. So a process must be created by another process. In response to the CREATE call, the Operating System obtains a new PCB from the pool of free memory, fills the fields with provided and/or default parameters, and inserts the PCB into the ready list-thus making the specified process eligible to run. Some of the parameters definable at the process-creation time include: (a) Level of privilege, such as system or user (b) Priority (c) Size and memory requirements (d) Maximum data area and/or stack size (e) Memory protection information and access rights (f) Other system-dependent data

Typical error returns, implying that the process was not created as a result of this call, include: wrongID (illegal, or process already active), no space for PCB (usually transient; the call may be retries later), and calling process not authorized to invoke this function.

DELETE (process ID);

DELETE invocation causes the Operating System to destroy the designated process and remove it from the system. A process may delete itself or another process. The Operating System reacts by reclaiming all resources allocated to the specified process, closing files opened by or for the process, and performing whatever other housekeeping is necessary. Following this process, the PCB is removed from its place of residence in the list and is returned to the free pool. This makes the designated process dormant. The DELETE service is normally invoked as a part of orderly program termination.

To relieve users of the burden and to enhance probability of programs across different environments, many compilers compile the last END statement of a main program into a DELETE system call.

Almost all multiprogramming operating systems allow processes to terminate themselves, provided none of their spawned processes is active. Operating System designers differ in their attitude toward allowing one process to terminate

others. The issue here is none of convenience and efficiency versus system integrity. Allowing uncontrolled use of this function provides a malfunctioning or a malevolent process with the means of wiping out all other processes in the system. On the other hand, terminating a hierarchy of processes in a strictly guarded system where each process can only delete itself, and where the parent must wait for children to terminate first, could be a lengthy operation indeed. The usual compromise is to permit deletion of other processes but to restrict the range to the members of the family, to lower-priority processes only, or to some other subclass of processes.

Possible error returns from the DELETE call include: a child of this process is active (should terminate first), wrongID (the process does not exist), and calling process not authorized to invoke this function.

Abort (processID);

ABORT is a forced termination of a process. Although a process could conceivably abort itself, the most frequent use of this call is for involuntary terminations, such as removal of a malfunctioning process from the system. The Operating System performs much the same actions as in DELETE, except that it usually furnishes a register and memory dump, together with some information about the identity of the aborting process and the reason for the action. This information may be provided in a file, as a message on a terminal, or as an input to the system crash-dump analyzer utility. Obviously, the issue of restricting the authority to abort other processes, discussed in relation to the DELETE, is even more pronounced in relation to the ABORT call.

Error returns for ABORT are practically the same as those listed in the discussion of the DELETE call.

FORK/JOIN

Another method of process creation and termination is by means of the FORK/JOIN pair, originally introduced as primitives for multiprocessor systems. The FORK operation is used to split a sequence of instructions into two concurrently executable sequences. After reaching the identifier specified in FORK, a new process (child) is created to execute one branch of the forked code

while the creating (parent) process continues to execute the other. FORK usually returns the identity of the child to the parent process, and the parent can use that identifier to designate the identity of the child whose termination it wishes to await before invoking a JOIN operation. JOIN is used to merge the two sequences of code divided by the FORK, and it is available to a parent process for synchronization with a child.

The relationship between processes created by FORK is rather symbiotic in the sense that they execute from a single segment of code, and that a child usually initially obtains a copy of the variables of its parent.

SUSPEND (processKD);

The SUSPEND service is called SLEEP or BLOCK in some systems. The designated process is suspended indefinitely and placed in the suspended state. It does, however, remain in the system. A process may suspend itself or another process when authorized to do so by virtue of its level of privilege, priority, or family membership. When the running process suspends itself, it in effect voluntarily surrenders control to the operating system. The Operating System responds by inserting the target process's PCB into the suspended list and updating the PCB state field accordingly.

Suspending a suspended process usually has no effect, except in systems that keep track of the depth of suspension. In such systems, a process must be resumed at least as many times as if was suspended in order to become ready. To implement this feature, a suspend-count field has to be maintained in each PCB. Typical error returns include: process already suspended, wrongID, and caller not authorized.

RESUME (processID)

The RESUME service is called WAKEUP is some systems. This call resumes the target process, which is presumably suspended. Obviously, a suspended process cannot resume itself, because a process must be running to have its Operating System call processed. So a suspended process depends on a partner process to issue the RESUME. The Operating System responds by inserting the target process's PCB into the ready list, with the state updated. In

systems that keep track of the depth of suspension, the Operating System first increments the suspend count, moving the PCB only when the count reaches zero.

The SUSPEND/RESUME mechanism is convenient for relatively primitive and unstructured form of inter-process synchronization. It is often used in systems that do not support exchange of signals. Error returns include: process already active, wrongID, and caller not authorized.

DELAY (processID, time);

The system call DELAY is also known as SLEEP. The target process is suspended for the duration of the specified time period. The time may be expressed in terms of system clock ticks that are system-dependent and not portable, or in standard time units such as seconds and minutes. A process may delay itself or, optionally, delay some other process.

The actions of the Operating System in handling this call depend on processing interrupts from the programmable interval timer. The timed delay is a very useful system call for implementing time-outs. In this application a process initiates an action and puts itself to sleep for the duration of the time-out. When the delay (time-out) expires, control is given back to the calling process, which tests the outcome of the initiated action. Two other varieties of timed delay are cyclic rescheduling of a process at given intervals (e.g,. running it once every 5 minutes) and time-of-day scheduling, where a process is run at a specific time of the day. Examples of the latter are printing a shift log in a process-control system when a new crew is scheduled to take over, and backing up a database at midnight.

The error returns include: illegal time interval or unit, wrongID, and called not authorized. In Ada, a task may delay itself for a number of system clock ticks (system-dependent) or for a specified time period using the pre-declared floating-point type TIME. The DELAY statement is used for this purpose.

GET_ATTRIBUTES (processID, attribute_set);

GET_ATTRIBUTES is an inquiry to which the Operating System responds by providing the current values of the process attributes, or their specified subset,

from the PCB. This is normally the only way for a process to find out what its current attributes are, because it neither knows where its PCB is nor can access the protected Operating System space where the PCBs are usually kept.

This call may be used to monitor the status of a process, its resource usage and accounting information, or other public data stored in a PCB. The error returns include: no such attribute, wrongID, and caller not authorized. In Ada, a task may examine the values of certain task attributes by means of reading the pre-declared task attribute variables, such as T'ACTIVE, T'CALLABLE, T'PRIORITY, and T'TERMINATED, where T is the identity of the target task.

CHANGE_PRIORITY (processID, new_priority);

CHANGE_PRIORITY is an instance of a more general SET_PROCESS_ATTRIBUTES system call. Obviously, this call is not implemented in systems where process priority is static.

Run-time modifications of a process's priority may be used to increase or decrease a process's ability to compete for system resources. The idea is that priority of a process should rise and fall according to the relative importance of its momentary activity, thus making scheduling more responsive to changes of the global system state. Low-priority processes may abuse this call, and processes competing with the Operating System itself may corrupt the whole system. For these reasons, the authority to increase priority is usually restricted to changes within a certain range. For example, maximum may be specified, or the process may not exceed its parent's or group priority. Although changing priorities of other processes could be useful, most implementations restrict the calling process to manipulate its own priority only.

The error returns include: caller not authorized for the requested change and wrong ID. In Ada, a task may change its own priority by calling the SET_PRIORITY procedure, which is pre-declared in the language.

2.2.3.2 System Programs

System can be viewed as a collection of system programs. Most system supplies a large collection of system programs to solve common problems and provide a

convenient environment for program development and execution. These system programs can be divided into following categories:

(a) **File manipulation:** These programs create, delete, copy, rename, print, dump and manipulate files and directories.

(b) **Status information:** Some programs need the date, time, available memory, disk space, number of users etc and then format the information and print it on the terminal or file or on some output device.

(c) **File modification:** A number of text editors are provided to create the file and manipulate their contents.

(d) **Programming language support:** A number of compilers, assemblers, and interpreters are provided for common programming languages such as C, Basic etc. with the operating systems.

(e) **Program loading and execution:** With operating system loaders and linkers are provided which are required to load and execute the program after their compilation. There are different types of loaders such as absolute loader, relocatable loader, overlay loaders, and linkage editors etc.

(f) **Application programs:** In addition to above a number of common programs provided with operating system are database systems, compilers-compiler, statistical analysis package, text formatters etc.

The most important system program for an operating system is its command interpreter. It is the program which reads and interprets the commands given by the user. This program is also known as control card interpreter or command line interpreter or the console command processor (in CP/M) or the shell (In Unix). Its function is simple: get the next command and execute it. The commands given to the command interpreter are implemented in two ways. In one approach the command interpreter itself contains the code to execute the command. So the number of commands that can be given determine the size of the command interpreter. An alternative approach implements all commands by special system programs. So the command interpreter merely uses the command to identify a file to be loaded into memory and executed. Thus a command *delete X* would

search for a file called delete, load it into the memory and pass it the parameter X. In this approach new commands can be easily added to the system by creating new files of the proper name. The command interpreter, which can now be quite small, need not be changed in order to add new commands.

2.3 Summary

Operating system provides an environment in which programs are executed. The bigger systems are organized in a hierarchical manner in which each layer provides some functionality. The virtual machine approach makes it possible to run different operating system on the same real machine.

Operating system provides a number of services. At the lowest level system calls allow a running program to make requests from the operating system directly. System calls can be grouped into three major categories: Process and job control, Device and file manipulation, and Information maintenance. At a higher level, the command interpreter provides a mechanism for a user to issue a request without needing to write a program. System programs can be divided into the categories File manipulation, Status information, File modification, Programming language support, Program loading and execution, and Application programs.

2.4 Keywords

1. **System calls:** They provide an interface between the process and the operating system and allow user-level processes to request some services from the operating system which process itself is not allowed to do.
2. **Virtual machine:** It makes it possible to run different operating system on the same real machine and allow the sharing of the underlying physical machine resources between different virtual machines, each running its own operating system.
3. **System Program:** System can be viewed as a collection of system programs that solve common problems and provide a convenient environment for program development and execution.

2.5. SELF ASSESMENT QUESTIONS (SAQ)

1. What is extended machine view? What are the advantages of hierarchical operating system structure? Explain.
2. Define system call? What are there different categories? Explain using suitable examples.
3. What do you understand by a virtual machine? What are the different advantages of it? Write a detailed note.
4. What do you understand by system programs? What are their different categories? Explain.
5. What do you understand by command interpreter? What are the functions performed by it? Discuss the two different approached to implement it.

2.6 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: 3	***Writer: Dr. Rakesh Kumar***
CPU Scheduling	***Vetter: Prof. Dharminder Kr.***

3.0 OBJECTIVE

The objective of this lesson is to make the students familiar with the various issues of CPU scheduling. After studying this lesson, they will be familiar with:

1. Process states and transitions.
2. Different types of scheduler
3. Scheduling criteria
4. Scheduling algorithms

3.1 INTRODUCTION

In nearly every computer, the most often requested resource is processor. Many computers have only one processor, so this processor must be shared via time-multiplexing among all the programs that need to execute on the computer. So processor management is an important function carried out by the operating system. Here we need to make an important distinction between a program and an executing program.

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. Roughly speaking, the process state is a snapshot of the machine at that time. At different times during the execution of a program (at different times in a process) different snapshots (different process states) will be observed.

The operating system is responsible for managing all the processes that are running on a computer. It allocates each process a certain amount of time to use

the processor. In addition, the operating system also allocates 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). In a real operating system, the waiting and ready states are implemented as queues, which hold the processes in these states.

The assignment of physical processors to processes allows processors to accomplish work. The problem of determining when processors should be assigned and to which processes, is called processor scheduling or CPU scheduling.

When more than one process is runable, the operating system must decide which one first. The part of the operating system concerned with this decision is called the scheduler, and algorithm it uses is called the scheduling algorithm. In operating system literature, the term “scheduling” refers to a set of policies and mechanisms built into the operating system that govern the order in which the work to be done by a computer system is completed. A scheduler is an Operating System module that selects the next job to be admitted into the system and the next process to run. The primary objective of scheduling is to optimize system performance in accordance with the criteria deemed most important by the system designers.

3.2 PRESENTATION OF CONTENTS

3.2.1 Definition of Process

3.2.2 Process States and Transitions

3.2.1 Definition of Process

The notion of process is central to the understanding of operating systems. There are quite a few definitions presented in the literature, but no "perfect" definition has yet appeared.

The term "process" was first used by the designers of the MULTICS in 1960's. Since then, the term process is used somewhat interchangeably with 'task' or 'job'. The process has been given many definitions for instance

- A program in Execution.
- An asynchronous activity.
- The 'animated sprit' of a procedure in execution.
- The entity to which processors are assigned.
- The 'dispatchable' unit.

As we can see from above that there is no universally agreed upon definition, but the definition "Program in Execution" seem to be most frequently used. Now that we agreed upon the definition of process, the question is "what is the relation

between process and program?" In the following discussion we point out some of the difference between process and program. Process is not the same as program rather a process is more than a program code. A process is an active entity as oppose to program which consider being a 'passive' entity. As we all know that a program is an algorithm expressed in some suitable notation, (e.g., programming language). Being a passive, a program is only a part of process. Process, on the other hand, includes:

- Current value of Program Counter (PC)
- Contents of the processors registers
- Value of the variables
- The Process Stack (SP) which typically contains temporary data such as subroutine parameter, return address, and temporary variables.
- A data section that contains global variables.

A process is the unit of work in a system.

In Process model, all software on the computer is organized into a number of sequential processes. A process includes PC, registers, and variables. Conceptually, each process has its own virtual CPU. In reality, the CPU switches back and forth among 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:

- Code for the program.
- Program's static data.
- Program's dynamic data.
- Program's procedure call stack.
- Contents of general purpose register.
- Contents of program counter (PC)
- Contents of program status word (PSW).
- Operating Systems resource in use.

A process goes through a series of discrete process states.

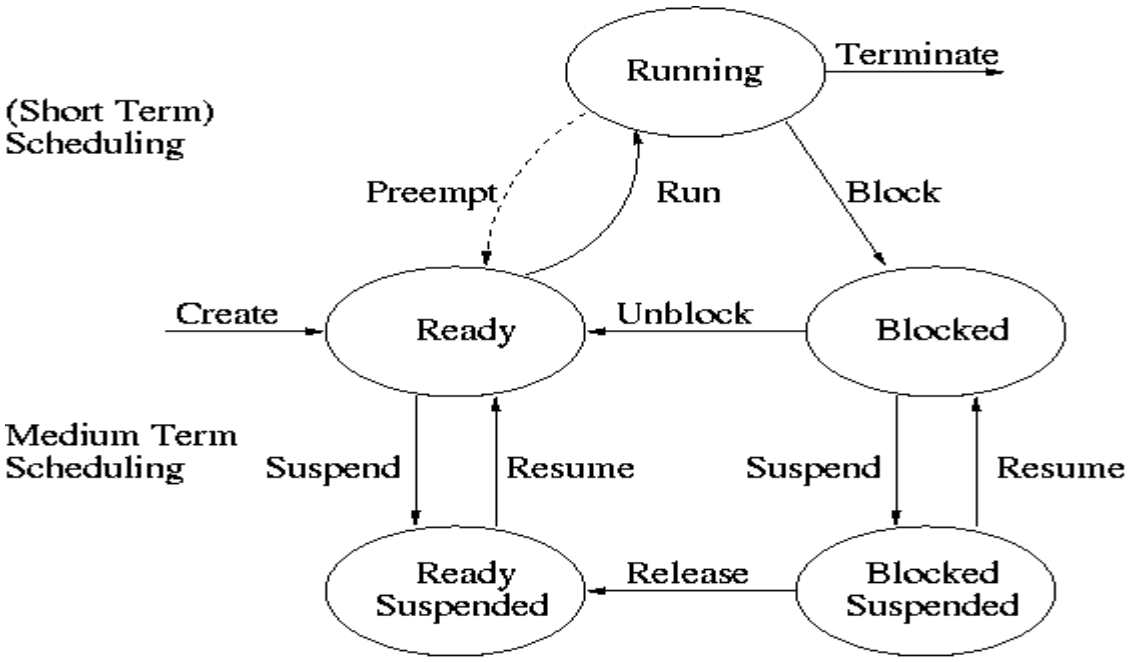
- **New State:** The process being created.

- **Running State:** A process is said to be running if it has the CPU, that is, process actually using the CPU at that particular instant.
- **Blocked (or waiting) State:** A process is said to be blocked if it is waiting for some event to happen such that as an I/O completion before it can proceed. Note that a process is unable to run until some external event happens.
- **Ready State:** A process is said to be ready if it use a CPU if one were available. A ready state process is runable but temporarily stopped running to let another process run.
- **Terminated state:** The process has finished execution.

3.2.2 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:

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

3.2.3 TYPES OF SCHEDULERS

Operating systems may feature up to 3 distinct types of schedulers: a long-term scheduler (also known as an admission scheduler or high-level scheduler), a mid-term or medium-term scheduler and a short-term scheduler (also known as a dispatcher). 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.

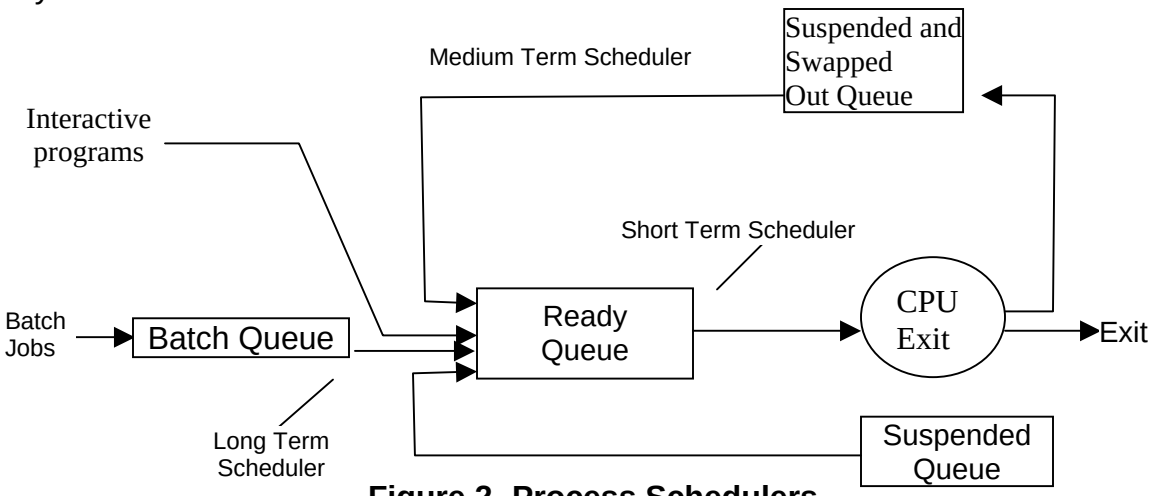


Figure 2- Process Schedulers

3.2.3.1 The long-term scheduler

The long-term scheduler decides when to start jobs, i.e., do not necessarily start them when submitted. CTSS (an early time sharing system at MIT) did this to insure decent interactive response time. The long-term scheduler, when present, works with the batch queue and selects the next batch job to be executed. Batch is usually reserved for resource-intensive (processor time, memory, special I/O devices), 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, such as memory size, expected execution time, and device requirements. 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. In a way, the long-term scheduler acts as a first-level throttle in keeping resource utilization at the desired level. For example, when the processor utilization is low, the scheduler may admit more jobs to increase the number of processes in a ready queue, and with it the probability of having some useful work awaiting processor allocation. Conversely, when the utilization factor becomes high as reflected in the response time, the long-term scheduler may opt to reduce the

rate of batch-job admission accordingly. 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. In terms of the process state-transition diagram, the long-term scheduler is basically in charge of the dormant-to-ready transitions. Ready processes are placed in the ready queue for consideration by the short-term scheduler.

3.2.3.2 The medium-term scheduler

The medium term scheduler suspend (swap 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 (pay more, get swapped out less) 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. Saving the image of a suspended process in secondary storage is called swapping and the process is said to be swapped out or rolled out.

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. To work properly, the medium-term scheduler must be provided with information about the memory requirements of swapped-out processes.

In terms of the state-transition diagram, 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 really 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. The responsibilities of the medium-term scheduler may be further complicated in some operating systems, in which some processes may request that their pages remain locked in physical memory:

3.2.3.3 The short-term scheduler

The long-term scheduler runs relatively infrequently, when a decision must be made as to the admission of new processes: maybe on average every ten seconds. The medium-term scheduler runs more frequently, deciding which process’s pages to swap to and from the swapping device: typically once a second. The short-term scheduler, often termed the dispatcher, executes most frequently (every few hundredths of a second) making fine-grained decisions as to which process to move to Running next. The short-term scheduler is invoked whenever an event occurs which provides the opportunity, or requires, the interruption of the current process and the new (or continued) execution of another process. Such opportunities include:

- Clock interrupts, provide the opportunity to reschedule every few milliseconds,
- Expected I/O interrupts, when previous I/O requests are finally satisfied,
- Operating system calls, when the running process asks the operating system to perform an activity on its behalf, and
- 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 (internal or external) 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. Some of the events occurred and, if so, to select the next process to be run.

Most of the process-management Operating System services discussed in this lesson 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.

3.2.4 SCHEDULING and PERFORMANCE CRITERIA

The objectives of a good scheduling policy include:

- Fairness.
- Efficiency.
- Low response time (important for interactive jobs).
- Low turnaround time (important for batch jobs).
- High throughput
- Repeatability.
- Fair across projects.
- Degrade gracefully under load.

The success of the short-term scheduler can be characterized by its success against user-oriented criteria under which a single user evaluates their perceived response, or system-oriented criteria where the focus is on efficient global use of resources such as the processor and memory. A common measure of the

system-oriented criteria is throughput, the rate at which tasks are completed. On a single-user, interactive operating system, and the user-oriented criteria take precedence: it is unlikely that an individual will exhaust resource consumption, but responsiveness remains all important. On a multi-user, multi-tasking system, the global system-oriented criteria are more important as they attempt to provide fair scheduling for all, subject to priorities and available resources.

3.2.4.1 User-oriented Scheduling Criteria

Response time

In an interactive system this measures the time between submissions of a new process request and the commencement of its execution. Alternatively, it can measure the time between a user issuing a request to interactive input (such as a prompt) and the time to echo the user's input or accept the carriage return.

Turnaround time

This is the time between submission of a new process and its completion. Depending on the mixture of current tasks, two submissions of identical processes will likely have different turnaround times. Turnaround time is the sum of execution and waiting times.

Deadlines

In a genuine real-time operating system, hard deadlines may be requested by processes. These either demands that the process is completed with a guaranteed upper-bound on its turnaround time, or provide a guarantee that the process will receive the processor in a guaranteed maximum time in the event of an interrupt. A real-time long-term scheduler should only accept a new process if it can guarantee required deadlines. In combination, the short-term scheduler must also meet these deadlines.

Predictability

With lower importance, users expect similar tasks to take similar times. Wild variations in response and turnaround times are distracting.

3.2.4.2 System-oriented Scheduling Criteria

Throughput

The short-term scheduler attempts to maximize the number of completed jobs per unit time. While this is constrained by the mixture of jobs, and their execution profiles, the policy affects utilization and thus completion.

Processor utilization

The percentage of time that the processor may be fed with work from Ready queue. In a single-user, interactive system, processor utilization is very unlikely to exceed a few percent.

Fairness

Subject to priorities, all processes should be treated fairly, and none should suffer processor starvation. This simply implies, in most cases, that all processes are moved to the ends of their respective state queues, and may not “jump the queue”.

Priorities

Conversely, when processes are assigned priorities, the scheduling policy should favor higher priorities.

3.2.5 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. One should intentionally avoid the word "optimization" because most scheduling algorithms actually implemented do not schedule optimally. They are based on heuristic techniques that yield good or near-optimal performance but rarely achieve absolutely optimal performance. The primary reason for this situation lies in the overhead that would be incurred by computing the optimal strategy at run-time, and by collecting the performance statistics necessary to perform the optimization. Of course, the optimization algorithms remain important, at least as a yardstick in evaluating the heuristics. 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. For example, a user experiencing 10-second response time to simple queries has little consolation in knowing that the system's average response time is under 2 seconds.

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. As is the case with most engineering problems, 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. For example, throughput and component utilization are the primary design objectives in a batch system. Multi-user systems are dominated by concerns regarding the terminal response time, and real-time operating systems are designed for the ability to handle burst of external events responsively.

3.2.7 SCHEDULING ALGORITHMS

The scheduling mechanisms described in this section may, at least in theory, be used by any of the three types of schedulers. As pointed out earlier, some algorithms are better suited to the needs of a particular type of scheduler. Depending on whether a particular scheduling discipline is primarily used by the long-term or by the short-term scheduler, we illustrate its working by using the term job or process for a unit of work, respectively.

The scheduling policies may be categorized as preemptive and non-preemptive. So it is important to distinguish preemptive from non-preemptive scheduling algorithms. Preemption means the operating system moves a process from running to ready without the process requesting it. Without preemption, the system implements “run to completion". Preemption needs a clock interrupt (or equivalent). 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

- It terminates, or
- It makes an I/O request which would block the process, or
- 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 new process arrives (perhaps at a higher priority), or
- An interrupt or signal occurs, or
- 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. Following are some scheduling algorithms we will study: FCFS Scheduling, Round Robin Scheduling, SJF Scheduling, SRTN Scheduling, Priority Scheduling, Multilevel Queue Scheduling, and Multilevel Feedback Queue Scheduling.

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

The simplest selection function is the First-Come-First-Served (FCFS) scheduling policy. In it

1. The operating system kernel maintains all Ready processes in a single queue,
2. The process at the head of the queue is always selected to execute next,
3. The Running process runs to completion, unless it requests blocking I/O,
4. 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. Since there is no discrimination on the basis of the required service, short jobs may suffer considerable turnaround delays and waiting times when one or more long jobs are in the system. For example, consider a system with two jobs, J1 and J2, with total execution times of 20 and 2 time units, respectively. If they arrive shortly one after the other in the order J1-J2, the turnaround times are 20 and 22 time units, respectively (J2 must wait for J1 to complete), thus yielding an average of 21 time units. The corresponding waiting times are 0 and 20 unit, yielding an average of 10 time units. However, when the same two jobs arrive in the opposite order, J2-J1, the average turnaround time drops to 11, and the average waiting time is only 1 time unit.

Compute-bound processes are favored over I/O-bound processes.

We can measure the effect of FCFS by examining:

- The average turnaround time of each task (the sum of its waiting and running times), or
- The normalized turnaround time (the ratio of running to waiting times).

3.2.6.2 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'' (i.e., we don't even switch to another process on I/O). In this situation, SJF has the shortest average waiting time. Assume you have a schedule with a long job right before a short job. Now if we swap the two jobs, this decreases the wait for the short by the length of the long job and increases the wait of the long job by the length of the short job and this in turn decreases the total waiting time for these two. Hence decreases the total waiting for all jobs and hence decreases the average waiting time as well. So in this policy whenever a long job is right before a short job, we swap them and decrease the average waiting time. Thus the lowest average waiting time occurs when there

are no short jobs rights before long jobs. This is an example of priority scheduling. This scheduling policy can starve processes that require a long burst.

3.2.6.3 Shortest Remaining Time Next (SRTN) Scheduling

Shortest remaining time next is a scheduling discipline in which the next scheduling entity, a job or a 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 (batch or ready) to find the job or 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. Depending on the outcome, the running process may continue, or it may be preempted and replaced by the shortest-remaining-time process. If preempted, the running process joins the ready queue.

SRTN is a provably optimal scheduling discipline in terms of minimizing the average waiting time of a given workload. SRTN scheduling is done in a consistent and predictable manner, 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. Preferred treatment of short jobs in SRTN tends to result in increased waiting times of long jobs in comparison with FCFS scheduling, but this is usually acceptable.

The SRTN discipline schedules optimally assuming that the exact future execution times of jobs or processes are known at the time of scheduling. In the case of short-term scheduling and preemption’s, even more detailed knowledge of the duration of each individual processor burst is required. Dependence on

future knowledge tends to limit the effectiveness of SRTN implementations in practice, because future process behavior is unknown in general and difficult to estimate reliably, except for some very specialized deterministic cases.

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.

SRTN scheduling has important theoretical implications, and it can serve as a yardstick for assessing performance of other, realizable scheduling disciplines in terms of their deviation from the optimum. Its practical application depends on the accuracy of prediction of the job and process behavior, with increased accuracy calling for more sophisticated methods and thus resulting in greater overhead. The preemptive variety of SRTN incurs the additional overhead of

frequent process switching and scheduler invocation to examine each and every process transition into the ready state. This work is wasted when the new ready process has a longer remaining execution time than the running process.

3.2.6.4 Round Robin

In interactive environments, such as time-sharing systems, the primary requirement is to provide reasonably good response time and, in general, 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).

It is a preemptive scheduling policy. This scheduling policy gives each process a slice of time (i.e., one quantum) before being preempted. As each process becomes ready, it joins the ready queue. A clock interrupt is generated at periodic intervals. When the interrupt occurs, the currently running process is preempted, and the oldest process in the ready queue is selected to run next. The time interval between each interrupt may vary.

It is one of the most common and most important scheduler. This is not the simplest scheduler, but it is the simplest preemptive scheduler. It works as follows:

- The processes that are ready to run (i.e. not blocked) are kept in a FIFO queue, called the "Ready" queue.
- There is a fixed time quantum (50 msec is a typical number) which is the maximum length that any process runs at a time.
- The currently active process P runs until one of two things happens:
  - P blocks (e.g. waiting for input). In that case, P is taken off the ready queue; it is in the "blocked" state.
  - P exhausts its time quantum. In this case, P is pre-empted, even though it is still able to run. It is put at the end of the ready queue.

    In either case, the process at the head of the ready queue is now made the active process.
- When a process unblocks (e.g. the input it's waiting for is complete) it is put at the end of the ready queue.

Suppose the time quantum is 50 msec, process P is executing, and it blocks after 20 msec. When it unblocks, and gets through the ready queue, it gets the standard 50 msec again; it doesn't somehow "save" the 30 msec that it missed last time.

It is an important preemptive scheduling policy and is essentially the preemptive version of FCFS. The key parameter here is the quantum size q. When a process is put into the running state a timer is set to q. If the timer goes off and the process is still running, the Operating System preempts the process. This process is moved to the ready state where it is placed at the rear of the ready queue. The process at the front of the ready list is removed from the ready list and run (i.e., moves to state running). When a process is created, it is placed at the rear of the ready list. As q gets large, RR approaches FCFS. As q gets small, RR approaches PS (Processor Sharing).

What value of q should we choose? Actually it 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.

Implementation of round robin scheduling requires support of an interval timer-preferably a dedicated one, as opposed to sharing the system time base. The

timer is usually set to interrupt the operating system whenever a time slice expires and thus force the scheduler to be invoked. The scheduler itself simply stores the context of the running process, moves it to the end of the ready queue, and dispatches the process at the head of the ready queue. The scheduler is also invoked to dispatch a new process whenever the running process surrenders control to the operating system before expiration of its time quantum, say, by requesting I/O. The interval timer is usually reset at that point, in order to provide the full time slot to the new running process. The frequent setting and resetting of a dedicated interval timer makes hardware support desirable in systems that use time slicing.

Round robin scheduling is often regarded as a "fair" scheduling discipline. It is also one of the best-known scheduling disciplines for achieving good and relatively evenly distributed terminal response time. The performance of round robin scheduling is very sensitive to the choice of the time slice. For this reason, duration of the time slice is often made user-tunable by means of the system generation process.

The relationship between the time slice and performance is markedly nonlinear. Reduction of the time slice should not be carried too far in anticipation of better response time. Too short a time slice may result in significant overhead due to the frequent timer interrupts and process switches. On the other hand, too long a time slice reduces the preemption overhead but increases response time.

Too short a time slice results in excessive overhead, and too long a time slice degenerates from round-robin to FCFS scheduling, as processes surrender control to the Operating System rather than being preempted by the interval timer. The "optimal" value of the time slice lies somewhere in between, but it is both system-dependent and workload-dependent. For example, the best value of time slice for our example may not turn out to be so good when other processes with different behavior are introduced in the system, that is, when characteristics of the workload change. This, unfortunately, is commonly the case with time-sharing systems where different types of programs may be submitted at different times.

In summary, round robin is primarily used in time-sharing and multi-user systems where terminal response time is important. Round robin scheduling generally discriminates against long non-interactive jobs and depends on the judicious choice of time slice for adequate performance. Duration of a time slice is a tunable system parameter that may be changed during system generation.

Variants of Round Robin

State dependent RR

It is same as RR but q is varied dynamically depending on the state of the system. It favors processes holding important resources. For example, non-swappable memory.

External priorities

In it a user can pay more and get bigger q. That is one process can be given a higher priority than another. But this is not an absolute priority, i.e., the lower priority (i.e., less important) process does get to run, but not as much as the high priority process.

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

In it each job is assigned a priority (externally, perhaps by charging more for higher priority) and the highest priority ready job is run. In this policy, If many processes have the highest priority, it uses RR among them. In principle, each process in the system is assigned a priority level, and the scheduler always chooses the highest-priority ready process. 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 priority-based scheduling is the possibility that low-priority processes may be effectively locked out by the higher priority ones. In general, completion of a process within finite time of its creation cannot be

guaranteed with this scheduling policy. In systems where such uncertainty cannot be tolerated, the usually remedy is provided by the aging priority, in which the priority of each process is gradually increased after the process spends a certain amount of time in the system. Eventually, the older processes attain high priority and are ensured of completion in finite time.

By means of assigning priorities to processes, system programmers can influence the order in which an ED scheduler services coincident external events. However, the high-priority ones may starve low-priority processes. Since it gives little consideration to resource requirements of processes, event-driven scheduling cannot be expected to excel in general-purpose systems, such as university computing centers, where a large number of user processes are run at the same (default) level of priority.

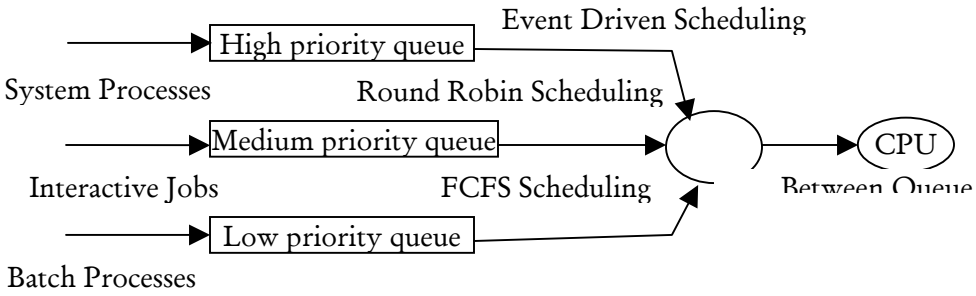
Another variant of priority-based scheduling is used in the so-called hard real-time systems, where each process must be guaranteed execution before expiration of its deadline. In such systems, time-critical processes are assumed to be assigned execution deadlines. The system workload consists of a combination 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 or the least slack 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. Interestingly, neither of these schedulers is optimal in multiprocessor environments.

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.

3.2.6.6 Multiple-Level Queues (MLQ) Scheduling

The scheduling policies discussed so far are more or less suited to particular applications, with potentially poor performance when applied inappropriately. What should one use in a mixed system, with some time-critical events, a multitude of interactive users, and some very long non-interactive jobs? One approach is to combine several scheduling disciplines. A mix of scheduling disciplines may best service a mixed environment, each charged with what it does best. 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.



Multilevel Queue Scheduling

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. This approach is often called multiple-level queues (MLQ) scheduling. A division of the workload might be into system processes, interactive programs, and batch jobs. This would result in three ready queues, as depicted in above Figure. A process may be assigned to a specific queue on the basis of its attributes, which may be user-or system-supplied. 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 (e.g. system processes) 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 (e.g., RR for interactive processes). 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.

Multiple queues scheduling is a very general discipline that combines the advantages of the "pure" mechanisms discussed earlier. MLQ scheduling may also impose the combined overhead of its constituent scheduling disciplines. However, assigning classes of processes that a particular discipline handles poorly by itself to a more appropriate queue may offset the worst-case behavior of each individual discipline. Potential advantages of MLQ were recognized early on by the O/S designers who have employed it in the so-called fore-ground/background (F/B) system. An F/B system, in its usual form, uses a two-level queue-scheduling discipline. The workload of the system is divided into two queues-a high-priority queue of interactive and time-critical processes and other processes that do not service external events. The foreground queue is serviced in the event-driven manner, and it can preempt processes executing in the background.

3.2.6.7 Multiple-Level Queues with Feedback Scheduling

Multiple queues in a system may be used to increase the effectiveness and adaptive ness of scheduling in the form of multiple-level queues with feedback. Rather than having fixed classes of processes allocated to specific queues, the idea is to make traversal of a process through the system dependent on its run-time behavior. For example, each process may start at the top-level queue. If the process is completed within a given time slice, it departs the system after having received the royal treatment. Processes that need more than one time slice may

be reassigned by the operating system to a lower-priority queue, which gets a lower percentage of the processor time. If the process is still now 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. This philosophy is supported by program-behavior research findings suggesting that completion rate has a tendency to decrease with attained service. In other words, the more service a process receives, the less likely it is to complete if given a little more service. Thus the feedback in MLQ mechanisms tends to rank the processes dynamically according to the observed amount of attained service, with a preference for those that have received less.

On the other hand, if a process surrenders control to the OS before its time slice expires, being moved up in the hierarchy of queues may reward it. As before, different queues may be serviced using different scheduling discipline. In contrast to the ordinary multiple-level queues, the introduction of feedback makes scheduling adaptive and responsive to the actual, measured run-time behavior of processes, as opposed to the fixed classification that may be defeated by incorrect guessing or abuse of authority. A multiple-level queue with feedback is the most general scheduling discipline that may incorporate any or all of the simple scheduling strategies discussed earlier. Its overhead may also combine the elements of each constituent scheduler, in addition to the overhead imposed by the global queue manipulation and the process-behavior monitoring necessary to implement this scheduling discipline.

3.3 SUMMARY

An important, although rarely explicit, function of process management is processor allocation. 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.

3.4 Keywords

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: the decisions as to which (Ready) process to execute next.

Non-preemptive scheduling: In non-preemptive scheduling, 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 preemptive scheduling, 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.

3.5 SELF-ASSESSMENT QUESTIONS (SAQ)

1. Discuss various process scheduling policies with their cons and pros.
2. Define process. What is the difference between a process and a program? Explain.
3. What are the different states of a process? Explain using a process state transition diagram.
4. Which type of scheduling is used in real life operating systems? Why?
5. Which action should the short-term scheduler take when it is invoked but no process is in the ready state? Is this situation possible?
6. How can we compare performance of various scheduling policies before actually implementing them in an operating system?
7. Shortest Job First (SJF) is a sort of priority scheduling. Comment.

8. What do you understand by starvation? How does SJF cause starvation? What is the solution of this problem?
9. What qualities are to be there in a scheduling policy? Explain.
10. Differentiate between user-oriented scheduling criteria and system-oriented scheduling criteria.

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.

Lesson number: 4	***Writer: Dr. Rakesh Kumar***
Deadlocks	**Vetter:** ***Prof. Dharminder Kr.***

Crises and deadlocks when they occur have at least this advantage that they force us to think."- Jawaharlal Nehru (1889 - 1964)

4.0 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

4.1 Introduction

We can understand the notion of a deadlock from the following simple real-life example. To be able to write a letter one needs a letter pad and a pen. Suppose there in one letter pad and one pen on a table with two persons seated around the table, Mr. A and Ms. B. Both Mr. A and Ms. B are desirous of writing a letter. So both try to acquire the resources they need. Suppose Mr. A was able to get the letter pad. In the meantime, Ms. B was able to grab the pen. Note that each of them has one of the two resources they need to proceed to write a letter. If they hold on to the resource they possess and await the release of the resource by the other, then neither of them can proceed. They are deadlocked.

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.

4.2 Presentation of contents

4.2.1 Definition

4.2.2 Preemptable and Nonpreemptable Resources

4.2.3 Necessary and Sufficient Deadlock Conditions

4.2.4 Resource-Allocation Graph

- 4.2.4.1 Interpreting a Resource Allocation Graph with Single Resource Instances

4.2.5 Dealing with Deadlock

4.2.6 Deadlock Prevention

- 4.2.6.1 Elimination of “Mutual Exclusion” Condition
- 4.2.6.2 Elimination of “Hold and Wait” Condition
- 4.2.6.3 Elimination of “No-preemption” Condition
- 4.2.6.4 Elimination of “Circular Wait” Condition

4.2.7 Deadlock Avoidance

- 4.2.7.1 Banker’s Algorithm

4.2.1 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.

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. 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. It is important to note that the number of processes and the number and kind of resources possessed and requested are unimportant.

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.

The simplest example of deadlock is where process 1 has been allocated non-shareable resources A, say, a tap drive, and process 2 has be allocated non-sharable resource B, say, a printer. Now, if it turns out that process 1 needs resource B (printer) to proceed and process 2 needs resource A (the tape drive) to proceed and these are the only two processes in the system, each has blocked the other and all useful work in the system stops. This situation is termed as deadlock. The system is in deadlock state because each process holds a resource being requested by the other process and neither process is willing to release the resource it holds.

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.

4.2.2 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.

4.2.3 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.

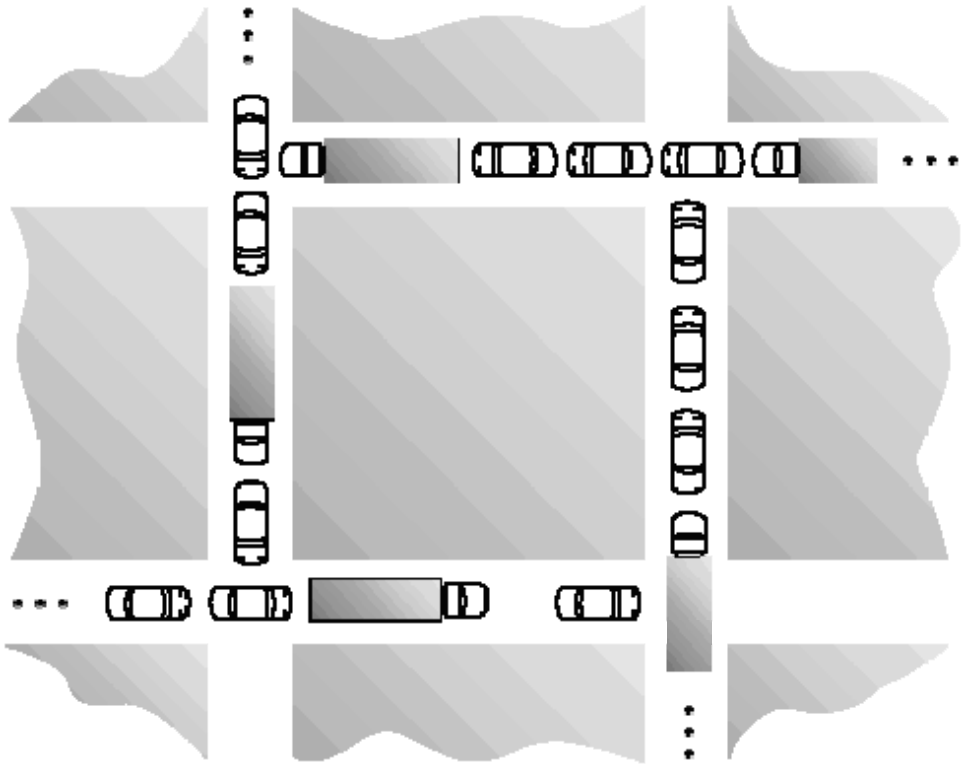
4. Circular Wait Condition

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.

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.

4.2.4 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.

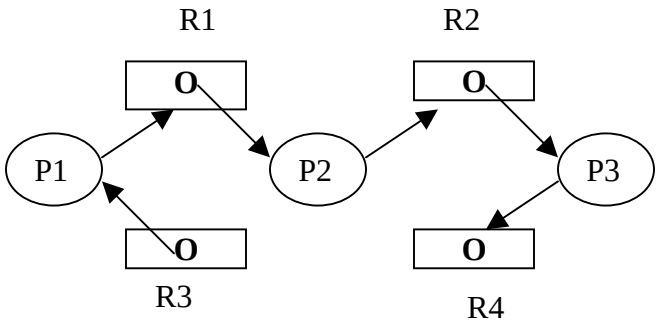
When a request is made, a request edge is added.

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

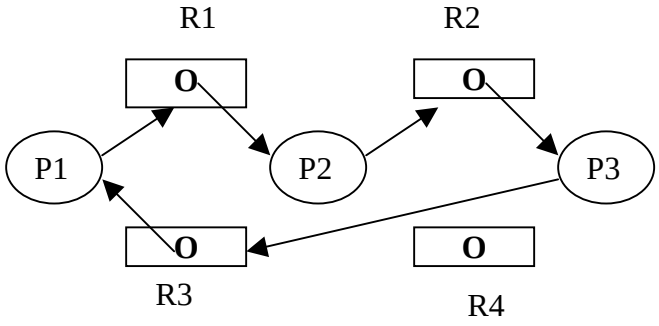
When process releases the resource, the assignment edge is deleted.

4.2.4.1 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.

4.2.5 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

4.2.6 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.

4.2.6.1 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.

4.2.6.2 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.

4.2.6.3 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.

4.2.6.4 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.

4.2.7 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.

4.2.7.1 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	Available

B	2	5	Units = 1
C	2	4	
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.

4.2.7.2 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.

4.2.8 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.

4.2.9 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.

Sometimes, we can do better. For example, if we checkpoint a process from time to time, we can roll it back to the latest checkpoint, hopefully to a time before it grabbed the resource in question. 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 OS, 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).

4.2.10 Mixed approaches to deadlock handling

The deadlock handling approaches differ in terms of theirv 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.

4.2.11 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

4.3 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

4.4 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

4.5 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?

4.6 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: 5 ***Writer: Dr. Rakesh Kumar***

Contiguous Memory Management - I ***Vetter: Prof. Dharminder Kumar***

5.0 OBJECTIVE

The lesson presents the principles of managing the main memory, one of the most precious resources in a multiprogramming system. In our sample hierarchy of Operating System layers, memory management belongs to layer 3. Memory management is primarily concerned with allocation of physical memory of finite capacity to requesting processes. No process may be activated before a certain amount of memory can be allocated to it. The objective of this lesson is to make the students acquainted with the concepts of contiguous memory management.

5.1 INTRODUCTION

Memory is large array of words or bytes, each having its unique address. CPU fetches instructions from memory according to value of program counter. The instructions undergo instruction execution cycle. To increase both CPU utilization and speed of its response to users, computers must keep several processes in memory. Specifically, the memory management modules are concerned with following four functions:

1. Keeping track of whether each location is allocated or unallocated, to which process and how much.
2. Deciding to whom the memory is allocated, how much, when and where. If memory is to be shared by more than one process concurrently, it must be determined which process' request should be satisfied.
3. Once it is decided to allocate memory, the specific locations must be selected and allocated. Memory status information is updated.
4. 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.

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

These systems can be divided into two major parts (i) Contiguous and (ii) Non-Contiguous

Contiguous Memory Management: In this approach, each program occupies a single contiguous block of storage locations.

Non-Contiguous Memory Management: In these, a program is divided into several blocks or segments that may be placed throughout main storage in pieces or chunks not necessarily adjacent to one another. It is the function of Operating System to manage these different chunks in such a way that they appear to be contiguous to the user.

Various issues to be considered in various memory management schemes are relocation, address translation, protection, sharing, and evaluation.

Relocation and address translation: The process of associating program instructions and data to physical memory addresses is called address binding or relocation. So binding is mapping from one address to another. It is of two types:

- Static Binding: It is taking place before execution; it may be (i) Compile time: where the compiler or assembler translates symbolic addresses to absolute addresses and (ii) Load time where the compiler translates symbolic addresses to relative addresses. The loader translates these to absolute addresses.

- Dynamic Binding: In it new locations are determined during execution. The program retains its relative addresses. The absolute addresses are generated by hardware.

Memory Protection and Sharing: Protection is used to avoid interference between programs existing in memory. Sharing is the opposite of protection.

Evaluation: Evaluation of these schemes is done on various parameters such as:

- **Wasted memory:** It is the amount of physical memory, which remains unused and thus wasted.
- **Access time** is the time to access the physical memory by the Operating System.
- **Time complexity** is related to overheads of the allocation or deallocation methods.

5.2 PRESENTATION OF CONTENTS

5.2.1 Single Contiguous Memory Management

5.2.2 Fixed Partitioned Memory Management System

5.2.2.1 Principles of Operation

5.2.2.2 Fragmentation

5.2.2.3 Swapping

5.2.2.4 Relocation

5.2.2.4.1 Static Relocation

5.2.2.4.2 Dynamic Relocation

5.2.2.5 Protection

5.2.2.6 Sharing

5.2.2.7 Evaluation

5.2.1 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 Operating System. Mostly, the Operating System resides in low memory (0 to P as shown in Figure 1). The remaining memory is allocated to transient or 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.

In this scheme, the starting physical address of the program is known at the time of compilation. The machine contains absolute addresses. They do not need to be changed or translated at the time of execution. So there is no issue of relocation or address translation.

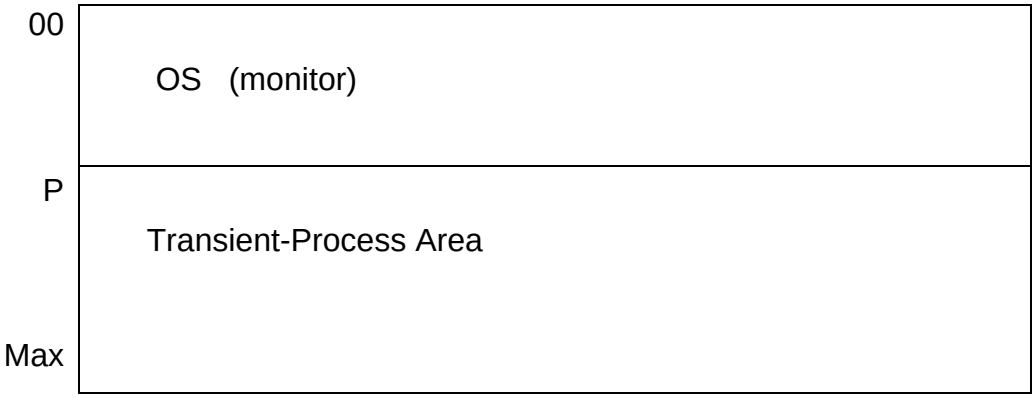


Figure 1. Single contiguous memory management

In this scheme as there is at most one process is in memory at any given time so there is a rare issue of interference between programs. However, it is desirable to protect the Operating System code from being tampered by the executing transient process.

A common way used in embedded systems to protect the Operating System code from user programs is to place the Operating System in read-only memory. This method is rarely used because of its inflexibility and inability to patch and update the Operating System code. In systems where the Operating System is in read-write memory, protection from user processes usually requires some sort of hardware assistance such as the fence registers and protection bits.

Fence registers are used to draw a boundary between the Operating System and the transient-process area. Assuming that the resident portion of the Operating System is in low memory, the fence register is set to the highest address occupied by Operating System code. Each memory address generated by a user process is compared against the fence. Any attempt to read or write the space below the fence may thus be detected and denied before completion of the related memory reference. Such violations usually trap to the Operating System,

which in turn may abort the offending program. To serve the purpose of protection, modification of the fence register must be a privileged operation not executable by user processes. Consequently, this method requires the hardware ability to distinguish between execution of the Operating System and of user processes, such as the one provided by user and supervisor modes of operation.

Another approach to memory protection is to record the access rights in the memory itself. One possibility is to associate a protection bit with each word in memory. The memory may then easily be divided into two zones of arbitrary size by setting all protection bits in one area, and resetting them in the other area. For example, initially all protection bits may be reset. During system startup, protection bits may be set in all locations where the Operating System is loaded. User programs may then be loaded and executed in the remaining memory locations. Prohibiting user processes from accessing any memory location whose protection bit is set may enforce Operating System protection. At the same time, the Operating System and system utilities, such as the loader, may be allowed unrestricted access to memory necessary for their activities. This approach requires a hardware-supported distinction between at least two distinct levels of privilege in the execution of machine instructions.

Sharing of code and data in memory does not make much sense in single-process environments, and single-process Operating System hardly ever support it. Users' programs may of course, pass data to each other in private arrangements, say, by means of memory locations known to be safe from being overwritten between executions of participating processes. Such schemes are obviously unreliable, and their use should be avoided whenever possible.

Single-process Operating System are relatively simple to design and to comprehend. They are often used in systems with little hardware support. But the lack of support for multiprogramming reduces utilization of both processor and memory. Processor cycles are wasted because there is no pending work that may be executed while the running process is waiting for completion of its I/O operations. Memory is underutilized because its portion not devoted to the Operating System and the single active user is wasted. On the average, wasted

memory in a specific system is related to the difference between the size of the transient-process area and the average process size weighted by the respective process-execution (and residence) times. This method has fast access time and very little time-complexity. Its usage is limited due to lack of multi-user facility.

One additional problem is sometimes encountered in systems with simplistic static forms of memory management. To be useable across a wide range of configurations with different capacities of installed memory, system programs in such environments tend to be designed to use the least amount of memory possible. Besides sacrificing speed and functionality, such programs usually take little advantage of additional memory when it is available.

5.2.2 FIXED PARTITIONED MEMORY MANGEMENT 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. System generation is a process of setting the Operating System to specific requirements. Various processes of the Operating System are allotted different partitions. 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. Since, in principle, at most one process may execute out of a given partition at any time, the number of partitions represents an upper limit on the number of active processes in a system i.e. degree of multiprogramming. Given the impact of memory partitioning on overall performance, some systems allow for manual redefinition of partition sizes.

Programs are queued to run in the smallest available partition. An executable prepared to run in one partition may not be able to run in another without being relinked. This technique is called absolute loading.

5.2.2.1 Principles of Operation

An example of partitioned memory is depicted in Figure 2. Out of the six partitions, one is assumed to be occupied by the resident portion of the OS, and three others by user processes P_i, P_j, and P_k, as indicated. The remaining two partitions, shaded in Figure 2, are free and available for allocation.

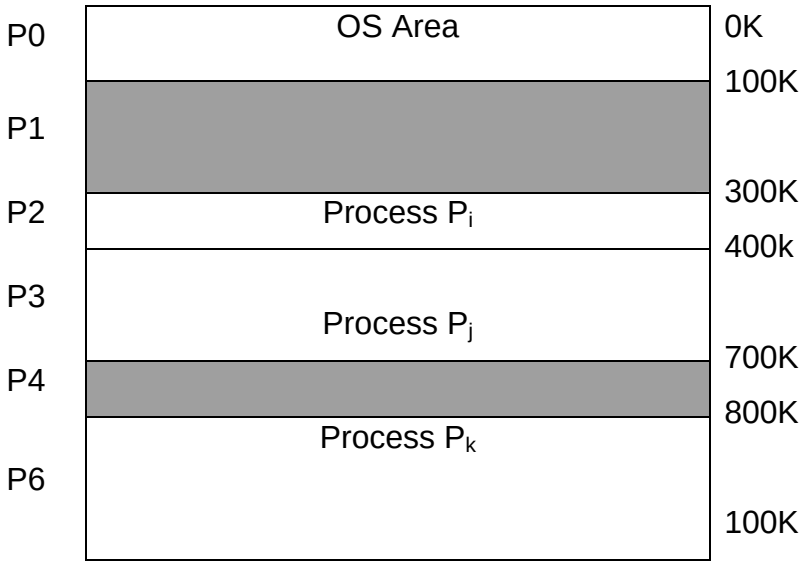


Figure 2 – Fixed Partitions

On declaring fixed partitions, the Operating System creates a Partition Description Table (PDT) to keep track of status of each partition for allocation purposes. A sample PDT format is given in Figure 3 according to the partitions given in Figure 2.

Partition Number	Partition Base	Partition size	Partition Status
0	0K	100K	Allocated
1	100K	200K	Free
2	300K	100K	Allocated
3	400K	300K	Allocated
4	700K	100K	Free
5	800K	200K	Allocated

Figure 3 – Partition description table

Each partition is described by its base address, size, and status. When fixed partitioning is used, only the status field of each entry varies i.e. free or allocated, in the course of system operation. Initially, all the entries are marked “FREE”. As

and when process is loaded into partitions, the status entry for that partition is changed to “ALLOCATED”.

Initially, all memory is available for user processes and is called hole. On arrival of a process, a hole large enough for that process is allocated to it. The Operating System then reads the program image from disk to the space reserved. After becoming resident in memory, the newly loaded process makes a transition to the ready state and thus becomes eligible for execution.

When a nonresident process is to be activated, the Operating System searches a free memory partition of sufficient size in the PDT. If the search is successful, the status field of the selected entry is marked ALLOCATED, and the process image is loaded into the corresponding partition. Since the assumed format of the PDT does not provide any indication as to which process is occupying a given partition, the identity of the assigned partition may be recorded in the PCB. When the process departs, using this information the status of related partition is made FREE. To implement these ideas, two questions are to be answered; (i) how to select a specific partition for a given process, (ii) what to do when no suitable partition is available for allocation. The three common strategies of partition allocation are:

(a) Best Fit

(b) First fit

(c) Worst Fit

Best-fit: This strategy allocates the smallest hole that is big enough to accommodate process. Entire list ordered by size is searched and matching smallest left over hole is chosen. For example, suppose a process requests 12KB of memory and the memory manager currently has a list of unallocated blocks of 6KB, 14KB, 19KB, 11KB, and 13KB blocks. The best-fit strategy will allocate 12KB of the 13KB block to the process.

First-fit: This strategy allocates the first available space that is big enough to accommodate process. 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. Using the same example as above, first fit will allocate 12KB of the 14KB block to the process.

Worst fit: This strategy allocates the largest hole. Entire list is searched. It chooses largest left over hole. Using the same example as above, worst fit will allocate 12KB of the 19KB block to the process, leaving a 7KB block for future use.

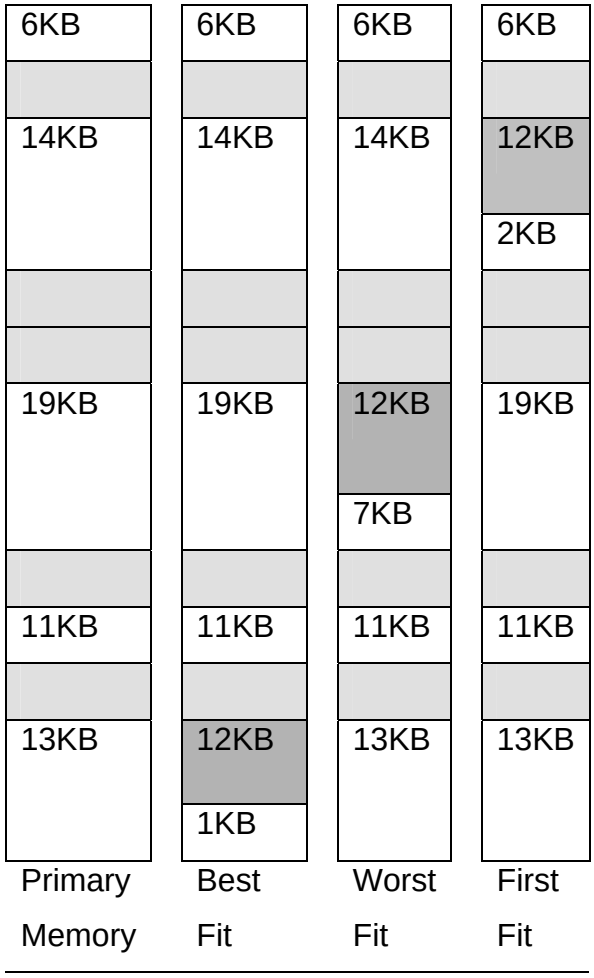


Figure 4

These strategies may be compared on the basis of execution speed and memory utilization. These algorithms have to search the PDT to identify a free partition of adequate size. However, while the first fit terminates upon finding the first such partition, the best fit must process all PDT entries to identify the tightest fit. So first fit tend to execute faster but best fit may achieve higher utilization of memory by creating the smallest possible gap resulting from the difference in size

between the process and its allocated partition. Both first-fit and best fit are better than worst-fit in terms of time and storage utilization, but first-fit is faster.

In case of a relatively small number of fixed partitions in a system, the execution time differences between these approaches may not be large enough to outweigh the lower degree of memory utilization attributable to the first fit. When the number of partitions is large neither first fit nor best fit is clearly superior.

Request for partitions may be 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. No free partition is large enough to accommodate the incoming process.
2. All partitions are allocated.
3. Some partitions are free, but none of them is large enough to accommodate the incoming process.

If the process to be created is too large to fit into any of the system partitions, the Operating System produces an error message. This is basically a configuration error that may be remedied by redefining the partitions accordingly. Another option is to reduce a program's memory requirements by recording and possibly using some sort of overlays.

The case when all partitions are allocated may be handled by deferring loading of the incoming process until a suitable partition can be allocated to it. An alternative is to force a memory-resident process to vacate a sufficiently large partition. Eviction to free the necessary space incurs the additional overhead of selecting a suitable victim and rolling it out to disk. This technique is called swapping. Both deferring and swapping are also applicable to handling the third case, where free but unsuitable partitions are available. If the deferring option is chosen, memory utilization may be kept high if the Operating System continues to allocate free partitions to other waiting processes with smaller memory requirements. However, doing so may violate the ordering of process activation’s intended by the scheduling algorithm and, in turn, affect performance of the system.

The described memory-allocation situations illustrate the close relationship and interaction between memory management and scheduling functions of the Operating System. Although the division of labor in actual systems may vary, the memory manager is generally charged with implementing memory allocation and replacement policies. Processor scheduling, on the other hand, determines which process gets the processor, when, and for how long. The short-term scheduler considers only the set of ready processes, that is, those that have all the needed resources except for the processor. Ready processes are, by definition, resident in memory. By influencing the membership of the set of resident processes, a memory manager may affect the scheduler's ability to perform. On the other hand, the effectiveness of the short-term scheduler influences the memory manager by affecting the average memory-residence times of processes.

In systems with fixed partitioning of memory, the number of partitions effectively sets an upper limit on the degree of multiprogramming. Within the confines of this limit, processor utilization may be improved by increasing the ratio of ready to resident processes. This may be accomplished by removing suspended processes from memory when otherwise ready ones are available for loading in the related partitions. A removed process is usually kept in secondary storage until all resources needed for its execution, except for memory and the processor may be allocated to it. At that point, the process in question becomes eligible for loading into the main memory. The medium-term scheduler and the memory manager cooperate in further processing of such processes.

The Operating System holds the processes waiting to be loaded in the memory in a queue. The two methods of maintaining this queue are (i) Multiple Queues and (ii) Single Queues.

Multiple Queues: In this method there are as many queues as the number of partitions. Separate queue for each partition is maintained in which processes are added as they arrive. When a process wants to occupy memory, it is added to a proper queue depending upon size of processes. Benefit of this method is that a small process is not loaded in large partition so as to avoid memory wastage. This leads to longer queue for small partitions.

Single Queue: In this method, there is only one queue for all ready processes. The order of processes in the queue depends on the scheduling algorithm. In this case, first fit allocation strategy is more efficient and fast.

5.2.2.2 Fragmentation

Some amount of memory is wasted both in single and multiple partition allocation techniques. Fragmentation 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. Selection of first fit and best fit can affect the amount of fragmentation. It is severe in variable size partitioning schemes. Compaction is a technique that is used to overcome this.

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 i.e. available memory internal to partition. It is severe in fixed partitioning schemes.

5.2.2.3 Swapping

Removing suspended or preempted processes from memory and their subsequent bringing back is called swapping. The basic idea of swapping is to treat main memory as a ‘pre-emptable’ resource. Lifting the program from the memory and placing it on the disk is called ‘Swapping out’. To bring the program again from the disk into the main memory is called ‘Swapping in’. Normally, a blocked process is swapped out so as to create available space for a ready process. This results in improving CPU utilization. Swapping has traditionally been used to implement multiprogramming in systems with restrictive memory capacity. Swapping may also be helpful for improving processor utilization in partitioned memory environments by increasing the ratio of ready to resident processes. Swapping is usually employed in memory-management systems with contiguous allocation, such as fixed and variable partitioned memory and

segmentation. Somewhat modified forms of swapping may also be present in virtual memory systems based on segmentation or on paging. Swapping brings flexibility even to systems with fixed partitions.

When the scheduler decides to admit a new process for which no suitable free partition can be found, the swapper may be invoked to vacate such a partition. The swapper is an Operating System process whose major responsibilities include:

- Selection of processes to swap out: Its criteria is suspended/blocked state, low priority, time spent in memory.
- Selection of processes to swap in: Its criteria are time spent on swapping device and priority.
- Allocation and management of swap space on a swapping device. Swap space can be system wide or dedicated.

Thus the swapper performs most of the functions of the medium-term scheduler. The swapper usually selects a victim among the suspended processes that occupy partitions large enough to satisfy the needs of the incoming process.

Although the mechanics of swapping out following the choice of a victim process is fairly simple in principle, implementation of swapping requires some specific provisions and considerations in Operating System that support it. These generally include the file system, specific Operating System services, and relocation.

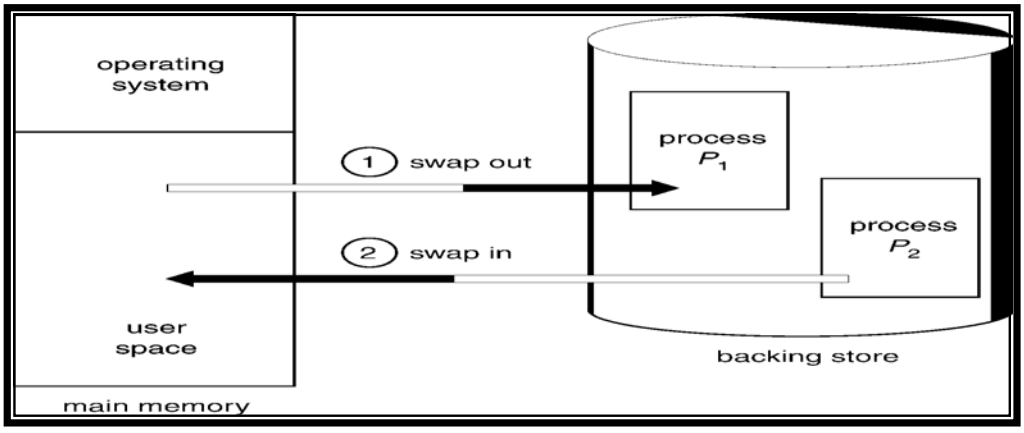


Figure 5 showing process of Swapping

A process is typically prepared for execution and submitted to the Operating System in the form of a file that contains a program in executable form and the related data. This file may also contain process attributes, such as priority and memory requirements. Such a file is sometimes called a process image. Since a process usually modifies its stack and data when executing, a partially executed process generally has a run-time image different from its initial static process image recorded on disk. Therefore, the dynamic run-time state of the process to be swapped out must be recorded for its proper subsequent resumption. In general, the modifiable portion of a process's state consists of the contents of its data and stack locations, as well as of the processor registers. Code is also subject to run-time modifications in systems that permit the code to modify itself. Therefore, the contents of a sizable portion or of the entire address space of a victim process must be copied to disk during the swapping-out operation. Since the static process image is used for initial activation, the (modified) run-time image should not overwrite the static process image on disk. Consequently, a separate swap file must be available for storing the dynamic image of a rolled-out process. There are two basic options regarding placement of a swap file:

- System-wide swap file
- Dedicated, per-process, swap files

In either case, swapping space for each swappable process is usually reserved and allocated statically, at process creation time, to avoid the overhead of this potentially lengthy operation at swap time.

In the system-wide swap file approach, a single large file is created, usually in the course of system initialization, to handle swapping requirements of all processes. The swap file is commonly placed on a fast secondary-storage device so as to reduce the latency of swapping. The location of each swapped out process image is noted within that file. An important trade-off in implementing a system-wide swap file is the choice of its size. If a smaller area is reserved for this file, the Operating System may not be able to swap out processes beyond a certain limit, thus affecting the performance.

An alternative is to have a dedicated swap file for each swappable process in the system. These swap files may be created either dynamically at process creation time or statically at program preparation time. This method is very flexible, but can be very inefficient due to the increased number of files and directories. In either case, the advantages of maintenance of separate swap files include elimination of the system swap-file dimensioning problem and of that file's overflow errors at run-time, and non-imposition of restrictions on the number of active processes. The disadvantages include more disk space expended on swapping, slower access, and more complicated addressing of swapping files scattered on the secondary storage.

Regardless of the type of swapping file used, the need to access secondary storage makes swapping a lengthy operation relative to processor instruction execution. This overhead must be taken into consideration in the decision of whether to swap a process in order to make room for another one.

Delays of this magnitude may be unacceptable for interrupt-service routines or other time-critical processes. For example, swapping out of a momentarily inactive terminal driver in a time-sharing system is certainly a questionable "optimization." Operating System that support swapping usually copes with this problem by providing some means for system programmers to declare a given process as being swappable or not. In effect, after the initial loading, an unswappable process remains fixed in memory even when it is temporarily suspended. Although this service is useful, a programmer may abuse it by declaring an excessive number of processes as fixed, thereby reducing the benefits of swapping. For this reason, the authority to designate a process as being un-swappable is usually restricted to a given class of privileged processes and users. All other processes, by default, may be treated as swappable.

An important issue in systems that support swapping is whether process-to-partition binding is static or dynamic, i.e., whether a swapped-out process can subsequently be loaded only into the specific partition from which it was removed or into any partition of adequate size. In general, static binding of processes to partitions may be done in any system with static partitioning of memory,

irrespective of whether swapping is supported or not. Static process-to-partition binding eliminates the run-time overhead of partition allocation at the expense of lower utilization of memory due to potentially unbalanced use of partitions. On the other hand, systems in which processes are not permanently bound to specific partitions are much more flexible and have a greater potential for efficient use of memory. The price paid for dynamic binding of processes to partitions is the overhead incurred by partition allocation whenever a new process or a swapped process is to be loaded into main memory. Moreover, dynamic allocation of partitions usually requires some sort of hardware support for dynamic relocation.

5.2.2.4 Relocation

The term program relocatability refers to the ability to load and execute a given program into an arbitrary place in memory. Since different load addresses may be assigned during different executions of a single relocatable program, a distinction is often made between virtual addresses (or logical address) and the physical addresses where the program and its data are stored in memory during a given execution. In reality, the program may be loaded at different memory locations, which are called physical addresses. The problem of relocation and address translation is to find a way to map virtual addresses onto physical addresses. Depending on when and how the mapping from the virtual address space to the physical address space takes place in a given relocation scheme, there are two basic types of relocation: (i) Static relocation and (ii) Dynamic relocation.

5.2.2.4.1 Static Relocation

Static relocation is performed before or during the loading of the program into memory, by a relocating linker/ loader. Constants, physical I/O port addresses, and offsets relative to the program counter are examples of values that are not location-sensitive and that do not need to be adjusted for relocation. Other forms of addresses of operands may depend on the location of a program in memory so must be adjusted accordingly when the program is being loaded or moved to a different area of memory.

A language translator typically prepares the object module by assuming the virtual address 0 to be the starting address of the program, thus making virtual addresses relative to the program loading address. Relocation information, including virtual addresses that need adjustment following determination of the physical load address, is provided for subsequent processing by the linker and loader. Either when the linker combines object modules or when the process image is being loaded, all program locations that need relocation are adjusted in accordance with the actual starting physical address allocated to the program. Once the program is in memory, values that need relocation are indistinguishable from those that do not.

Since relocation information in memory is usually lost following the loading, a partially executed statically relocatable program cannot be simply copied from one area of memory into another and be expected to continue to execute properly. In systems with static relocation a swapped-out process must either be swapped back into the same partition from which it was evicted, or software relocation must be repeated whenever the process is to be loaded into a different partition. Given the considerable space and time complexity of software relocation, systems with static relocation are practically restricted to supporting only static binding of processes to partitions. This method is slow process because it involves software translation. It is used only once before the initial loading of the program.

5.2.2.4.2 Dynamic Relocation

In it, mapping from the virtual address space to the physical address space is performed at run-time. Process images in systems with dynamic relocation are also prepared assuming the starting location to be a virtual address 0, and they are loaded in memory without any relocation adjustments. When the related process is being executed, all of its memory references are relocated during instruction execution before physical memory is actually accesses. This process is often implemented by means of specialized base registers. After allocating a suitable partition and loading a process image in memory, the Operating System sets a base register to the starting physical load address. This value is normally

obtained from the relevant entry of the PDT. Each memory reference generated by the executing process is mapped into the corresponding physical address by having the contents of the base register added to it.

Dynamic relocation is illustrated in Figure 6. A sample process image prepared with an assumed starting address of virtual address 0 is shown unchanged before and after being loaded in memory. In this particular example, it is assumed that address 1000 is allocated as the starting address for loading the process image. This base address is normally available from the corresponding entry of the PDT, which is reachable by means of the link to the allocated partition in the PCB. Whenever the process in question is scheduled to run, the base register is loaded with this value in the course of process switching.

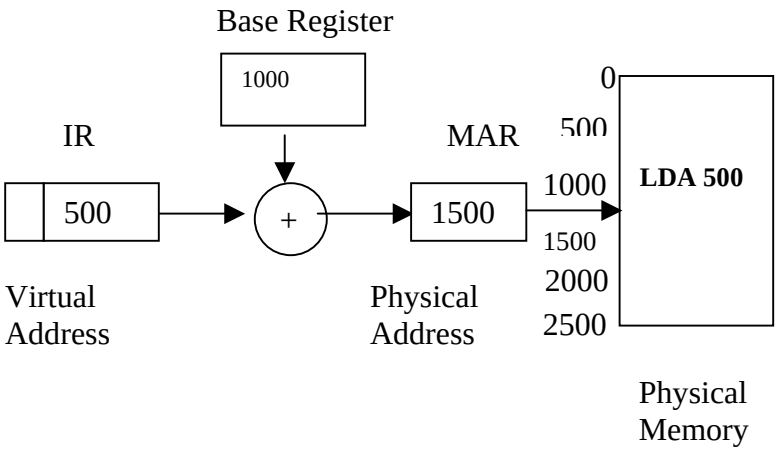


Figure 6 – Dynamic relocation

Relocation of memory references at run-time is illustrated by means of the instruction LDA 500, which is supposed to load the contents of the virtual address 500 (relative to program beginning) into the accumulator. As indicated, the target item actually resides at the physical address 1500 in memory. This address is produced by hardware by adding the contents of the base register to the virtual address given by the processor at run-time.

As suggested by Figure 5, relocation is performed by hardware and is invisible to programmers. In effect, all addresses in the process image are prepared by counting on the implicit based addressing to complete the relocation process at run-time. This approach makes a clear distinction between the virtual and the physical address space.

This is the most commonly used scheme amongst the schemes using fixed partitions due to its enhanced speed and flexibility. Its advantage is that it supports swapping easily. Only the base register value needs to be changed before dispatching.

5.2.2.5 Protection

Not only must the Operating System be protected from unauthorized tampering by user processes, but each user process must also be prevented from accessing the areas of memory allocated to other processes. Otherwise, a single erroneous or malevolent process may easily corrupt any or all other resident processes. There are two approaches for preventing such interference and achieving protection. These approaches involve the use of Limit Register and Protection Bits.

Implementation of memory protection in a given system tends to be greatly influenced by the available hardware support. In systems that use base registers for relocation, a common practice is to use limit registers for protection. The primary function of a limit register is to detect attempts to access address space beyond the boundary assigned to the executing program by the Operating System. The limit register is usually set to the highest virtual address in a program. As illustrated by Figure 6, each intended memory reference of an executing program is checked against the contents of the limit register before being forwarded to memory. In this way, any attempt to access a memory location outside of the specified area is detected and aborted by the protection hardware before being allowed to reach the memory. This violation usually traps to the Operating System, which may then take a remedial action, such as to terminate the offending process. The base and limit values for each process are normally kept in its PBC. Upon each process switch, the hardware base and limit registers are loaded with the values required for the new running process. Another approach to protection is to record the access rights in the memory itself. The bit-per-word approach described earlier, is not suitable for multiprogramming systems because it can separate only two distinct address spaces. Adding more bits to designate the identity of each word’s owner may solve this problem, but

this approach is rather costly. A more economical version of this idea has been implemented by associating a few individual words. For example, some models of the IBM 360 series use four such bits, called keys, per each 2 KB block of memory. When a process is loaded in memory, its identity is recorded in the protection bits of the occupied blocks. The validity of memory references is established at run-time by comparison of the running process's identity to the contents of protection bits of the memory block being accessed. If no match is found, the access is illegal and hardware traps to the Operating System for processing of the protection-violation exception. The Operating System is usually assigned a unique "master" key, say 0 that gives it unrestricted access to all blocks of memory. Note that this protection mechanism imposes certain restrictions on operating-system designers. For example, with 4-bit keys the maximum number of static partitions and of resident processes is 16. Likewise, associating protection bits with fixed-sized blocks forces partition sizes to be an integral number of such blocks.

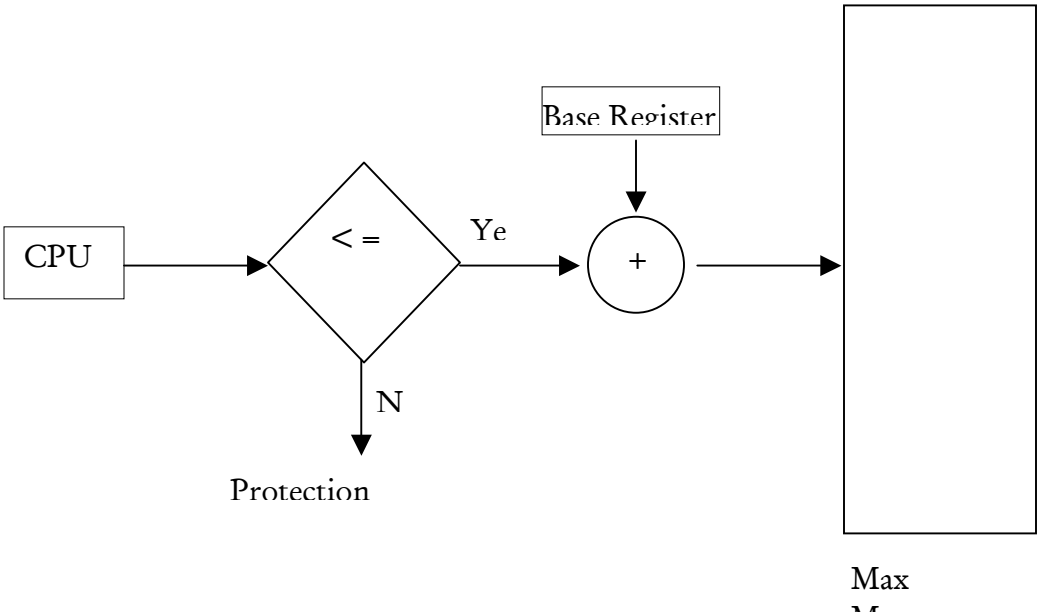


Figure 7 – Base-limit register

5.2.2.6 Sharing

Sharing of code and data poses a serious problem in fixed partitions because it might compromise on protection. There are three basic approaches to sharing in systems with fixed partitioning of memory:

- Entrust shared objects to Operating System.
- Maintain multiple copies, one per participating partition, of shared objects.
- Use shared memory partitions.

The easiest way to implement sharing without significantly compromising protection is to entrust shared objects to the Operating System. It means that any code or data goes through the Operating System for any request because the Operating System has the controlling access to shared resources. No additional provisions may be needed to support sharing. This scheme is possible but very tedious. Unfortunately, this simple approach increases the burden on the Operating System. Therefore, it is not followed in practice.

Unless objects are entrusted to the Operating System, sharing is quite difficult in systems with fixed partitioning of memory. The primary reason is their reliance on rather straightforward protection mechanisms based mostly on the strict isolation of distinct address spaces. Since memory partitions are fixed, disjoint, and usually difficult to access by processes not belonging to the Operating System, static partitioning of memory is not very conducive to sharing.

Another approach is to keep copies of the sharable code/ data in all partitions where required. It is wasteful and leads to inconsistencies. Since there is no commonly accessible original, each process runs using its copy of the shared object. Consequently, updates are made only to copies of the shared object. For consistency, updates made to any must be propagated to all other copies, by copying the shared data from the address space of the running process to all participating partitions upon every process switch. Swapping, when supported, introduces the additional complexity of potentially having one or more participating address spaces absent from main memory. This approach of sharing does not make much sense in view of the fact that no saving of memory may be expected.

Another traditional simple approach to sharing is to place the data in a dedicated "common" partition. However, any attempt by a participating process to access memory outside of its own partition is normally regarded as a protection violation. In systems with protection keys, changing the keys of all shared blocks upon every process switch in order to grant access rights to the currently running process may circumvent this obstacle. Keeping track of which blocks are shared and by whom, as well as the potentially frequent need to modify keys, results in notable Operating System overhead necessary to support this form of sharing. With base-limit registers, the use of shared partitions outside of-and potentially discontiguous to-the running process's partition requires some special provisions.

5.2.2.7 Evaluation

- Wasted memory: In fixed partitions, lot of memory is wasted due to both kinds of fragmentation.
- Access Time: Access time is not very high due to the assistance of special hardware. The translation from virtual address to physical address is done by hardware itself, thus enabling rapid access.
- Time complexity is very low because allocation/ deallocation routines are simple as the partitions are fixed.

5.3 Keywords

Contiguous Memory Management: In this approach, each program occupies a single contiguous block of storage locations.

First-fit: This allocates the first available space that is big enough to accommodate process.

Best-fit: This allocates the smallest hole that is big enough to accommodate process.

Worst fit: This strategy allocates the largest hole.

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.

Compaction is to shuffle memory contents and place all free memory together in one block.

Program relocatability refers to the ability to load and execute a given program into an arbitrary place in memory.

5.4 SUMMARY

In this lesson, we have presented Single Contiguous Memory Management and Fixed Partition Memory Management schemes for management of main memory that are characterized by contiguous allocation of memory. Single contiguous memory management is inefficient in terms of both CPU and memory utilization and does not support multiprogramming. All other schemes support multiprogramming by allowing address spaces of several processes to reside in main memory simultaneously. One approach is to statically divide the available physical memory into a number of fixed partitions and to satisfy requests for memory by granting suitable free partitions, if any. Fixed partition sizes limit the maximum allowable virtual-address space of any given process to the size of the largest partition (unless overlays are used). The total number of partitions in a given system limits the number of resident processes. Within the confines of this limit, the effectiveness of the short-term scheduler may be improved by employing swapping to increase the ratio of resident to ready processes. Systems with static partitioning suffer from internal fragmentation of memory.

Fixed partitioning of memory rely on hardware support for relocation and protection. Sharing is quite restrictive in these systems.

5.5 SELF ASSESSMENT QUESTIONS (SAQ)

1. What functions does a memory manager perform?
2. How is user address space loaded in one partition of memory protected from others?
3. What is the problem of fragmentation? How is it dealt with?
4. What do you understand by program relocatability? Differentiate between static and dynamic relocation.
5. Differentiate between first fit, best fit, and worst fit memory allocation strategies. Discuss their merits and demerits.

6. How is the tracks of status of memory is kept in partitioned memory management?
7. What do you mean by relocation of address space? What problems does it cause?
8. Differentiate between internal fragmentation and external fragmentation.
9. What is external fragmentation? What is compaction? What are the merits and demerits of compaction?
10. What are three basic approaches to sharing in systems with fixed partitioning of memory?

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, Godbole A.S., 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.

Lesson number: 6 ***Writer: Dr. Rakesh Kumar***

Contiguous Memory Management - II ***Vetter: Prof. Dharminder kUMAR***

6.0 OBJECTIVE

The main memory is the most precious resource in a multiprogramming system. There are a number of schemes to manage the main memory broadly categorize as contiguous and non-contiguous memory management schemes. Single contiguous memory allocation and fixed partition memory allocation schemes part of contiguous memory management have been discussed in previous lesson. The objective of this lesson is:

(i) To make the students familiar with the variable partitioned memory management scheme, a type of contiguous memory management scheme.

(ii) To explain the various memory compaction techniques.

(iii) To discuss the memory protection and sharing in variable partition memory management.

6.1 INTRODUCTION

As discussed in the last lesson, the memory management modules are concerned primarily with following functions: (a) Keeping track of whether each location is allocated or unallocated, to which process and how much, (b) deciding to whom the memory is allocated, how much, when and where, (c) once it is decided to allocate memory, the specific locations must be selected and allocated, and (d) handling the deallocation/reclamation of memory. The main problem with fixed partitioned memory management system discussed in the last lesson is determining the best region size to minimize the problem of internal and external 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. One possible solution to this problem is to allow the partitioning of the memory dynamically i.e. variable

partitioned memory management system. In this approach, partition sie and boundaries are changed during system operations to suit memory requirements of individual programs. Each job is allocated a partition whose size matches its memory requirements. Hence no internal fragmentation exists. However external fragmentation may arise due to existence of holes which are too small to accommodate any program.

6.2 PRESENTATION OF CONTENTS

6.2.1 Variable Partitioned Memory Allocation

- 6.2.1.1 Principles of Operation
- 6.2.1.2 Compaction
- 6.2.1.3 Buddy System
- 6.2.1.4 Protection
- 6.2.1.5 Sharing
- 6.2.1.6 Evaluation

6.2.1 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. Process is allocated exactly as much memory as required.

6.2.1.1 Principles of Operation

When instructed to load a process image, the memory-management module attempts to create a suitable partition for allocation to the process in question. The first step is to locate a contiguous free area of memory, which is equal to or larger than the process's size declared with the submitted image. If a suitable free area is found, the Operating System carves out a partition from it to provide an exact fit to the process's needs. The leftover chunk of free memory is returned to the pool of free memory for later use by the allocation module. The partition is created by entering its base, size, and status into the system Partition Description Table (PDT). A copy of, or some link to, this information is normally

recorded in the PCB. After loading the process image into the created partition, the process may be turned over to the Operating System module appropriate for its further processing, such as the short-term scheduler. If no suitable free area can be allocated, the Operating System returns an error indication.

When a resident process departs, Operating System returns the partition's space to the pool of free memory and invalidates the corresponding Partition Description Table entry. For swapped-out processes, the Operating System also invalidates the PCB field where the identity of the allocated partition is normally held.

The Operating System obviously needs to keep track of both partitions and free memory. Once created, a partition is defined by its base address and size. Those attributes remain essentially unchanged for as long as the related partition exists. In addition, for the purposes of process switching and swapping, it is important to know which partition belongs to a given process.

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 Operating System must keep track of the starting address and size of each free area of memory. This information may need to be updated each time a partition is created or terminated. 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. It is common to conserve space by building the free list within the free memory itself. For example, the first few words of each free area may be used to indicate the size of the area and to house a link to the successor area.

Figure 1 illustrates the working of variable partitioned memory. In this example, first Process 1, Process 2 and Process 3 are allocated memory as they arrive. When Process 2 is swapped out, the memory freed by Process 2 is available for any other process. So it is allocated to Process 4 and the size of partition for process 4 also varies. Again when process 2 arrives, it is allocated memory at different location that was freed by Process 1. Moreover, the size of partition also differs from the size of partition of process 1.From the given example, it is clear

that memory is allocated to processes as they arrive and on availability of memory. Partitions are created at the time of allocation according to size of process and not at the time of system generation.

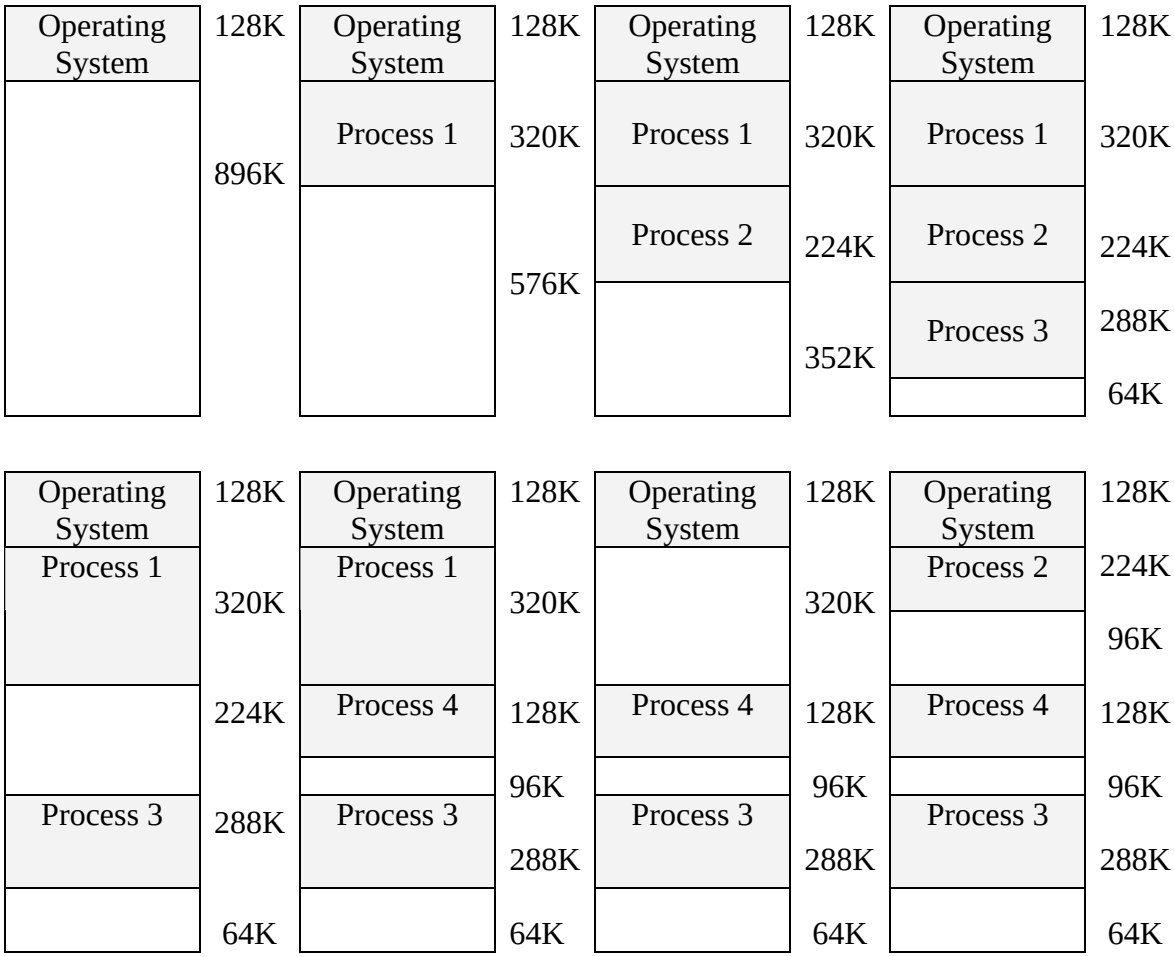


Figure 1– Partitions in dynamic memory partitioning

Common algorithms for selection of a free area of memory for creation of a partition (Step 1) are

(i) First fit

(ii) Best fit

(iii) Worst fit

(iv) Next Fit

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, as is

the case with first fit. 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. In general, next fit was not found to be superior to the first fit in reducing the amount of wasted memory.

First fit is generally faster because it terminates as soon as a free block large enough to house a new partition is found. Thus, on the average, first fit searches half of the free list per allocation. Best fit, on the other hand, searches the entire free list to find the smallest free block large enough to hold a partition being created. First fit is faster, but it does not minimize wasted memory for a given allocation. Best fit is slower, and it tends to produce small leftover free blocks that may be too small for subsequent allocations. However, when processing a series of request starting with an initially free memory, neither algorithm has been shown to be superior to the other in terms of wasted memory.

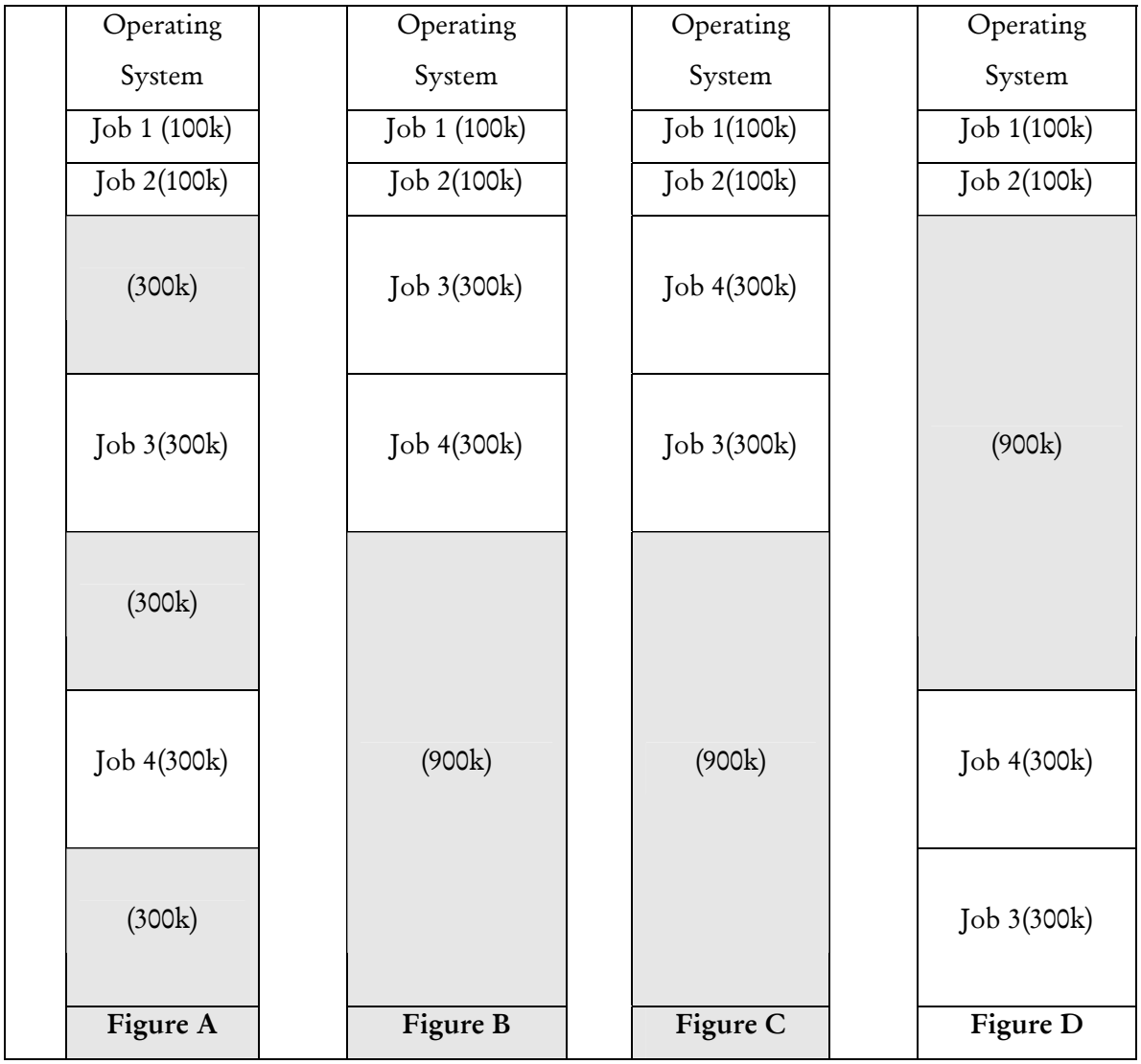
Worst fit is an antipode of best fit, as it allocates the largest free block, provided the block size exceeds the requested partition size. The idea behind the worst fit is to reduce the rate of production of small holes, which are quite common in best fit. However, some simulation studies indicate that worst fit allocation is not very effective in reducing wasted memory in the processing of a series of requests.

Termination of partitions in a system with dynamic allocation of memory may be performed by means of the procedure that recombines free areas, if possible, to reduce fragmentation of memory. When first fit or best fit is used, the free list may be sorted by address to facilitate recombination of free areas when partitions are deallocated.

6.2.1.2 Compaction

It is one solution to problem of external fragmentation. The goal here is to shuffle memory contents and place all free memory together in one block. Compaction is possible only if relocation is dynamic. This technique shifts the necessary process images to bring the free chunks of memory to adjacent positions to coalesce. Coalescing of adjacent free areas is a method often used to reduce fragmentation and the amount of wasted memory. However, such remedies tend to defer the impact of, rather than to prevent, the problem. The primary reason

for fragmentation is that, due to different lifetimes of resident objects, the pattern of returns of free areas is generally different from the order of allocations.



When memory becomes seriously fragmented, the only way out may be to relocate some or all partitions into one end of memory and thus combine the holes into one large free area. Since affected processes must be suspended and actually copied from one area of memory into another. It is important to decide when and how memory compaction is to be performed.

One simplest approach of doing compaction is to move all jobs towards one end of the memory and all holes in other direction resulting into one large hole of available memory. Now consider the memory allocation as shown above in figure

A. If we are using this simple approach we have to move Job 3 and Job 4 upward as shown in figure B, producing a large hole of 900 k after moving 2 blocks of 300k. But same can be achieved by moving Job 4 above the Job 3 as shown in figure C or moving Job 3 below Job 4 as in Figure D (Although in this case the large hole of available memory (900 k) is not at the end of the memory rather it is in the middle). So deciding an optimal compaction strategy is not an easy task.

Memory compaction may be performed whenever possible or only when needed. Some systems compact memory whenever a process departs, thus is collecting most of the free memory into a single large area. An alternative is to compact only upon a failure to allocate a suitable partition, provided that the combined size of free areas exceeds the needs of the request at hand.

Compaction involves a high overhead, but it increases the degree of multiprogramming. That is why; Operating System can accommodate a process with a larger size, which would have been impossible before compaction.

6.2.1.3. Buddy System:

This is another method of allocation-deallocation which speeds up merging of adjacent holes. This method facilitates merging of free space by allocating free areas with an affinity to recombine. It treats entire space available as a single block of $2^{k,}$ Requests for free areas are rounded up to the next integer power of base 2. When a free block of size 2^k is requested, the memory allocator attempts to satisfy it by allocating a free block from the list of free blocks of size 2^k. If none is available, the block of the next larger size, 2^{k+1}, is split in two halves (buddies) to satisfy the request. An important property of this allocation scheme is that the base address of the other buddy can be determined given the base address and size of one buddy (for a block of size 2^k, the two addresses differ only in the binary digit whose weight is 2^k). Thus, when a block is freed, a simple test of the status bit can reveal whether its buddy is also free. If so, the two blocks can be recombined to form the twice-larger original block. In addition to the free-list links, a status field is associated with each area of memory to indicate whether it is in use or not. Free blocks of equal size are often kept in separate free lists. Advantage of Buddy System is that it coalesces adjacent buffers or holes. Its

major disadvantage is that this method is very inefficient in terms of memory utilization.

As an example, consider a system with 1MB of memory (100000H) managed using the buddy allocation scheme. An initial request for a 192 KB block in such a system would require allocation of a 256 KB block (rounded up to the size that is a power of 2). Since no such block is initially available, the memory manager would form it by splitting the 1 MB block into two 512 KB buddies, and then splitting one of them to form two 256 KB blocks. The first split produces two 512 KB blocks (buddies) with starting addresses of 00000H (H stands for hexadecimal) and 80000H, respectively. The second split of the block at 80000H yields two 256 KB blocks (buddies) that start at 80000H and A0000H, respectively. Assume that the block starting at A0000H is allocated to the user. When the partition starting as A0000H is eventually terminated, the memory manager can identify the base address of its 256 KB buddy (buddies are of the same size) by toggling the address bit in the position that corresponds to the size of the block being released. In the presented example, 256 KB = 2^{18}, and toggling of that bit yields a 0 (in this example) in bit position 18, which, with all other bits unchanged, produces the address of the original buddy, 80000H. A quick inspection of the associated status word indicates whether the buddy at that address is free or not. If it is, the two buddies are coalesced to reform the 512 KB block starting at address 80000H, which was originally used to produce the smaller blocks to satisfy the pending request.

Example: The buddy memory allocation technique allocates memory in powers of 2, i.e 2^x, where x is an integer. Thus, the programmer has to decide on, or to write code to obtain, the upper limit of x. For instance, if the system had 2000K of physical memory, the upper limit on x would be 10, since 2^{10} (1024K) is the biggest allocatable block. This results in making it impossible to allocate everything in as a single chunk; the remaining 976K of memory would have to be taken in smaller blocks.

After deciding on the upper limit (let's call the upper limit *u*), the programmer has to decide on the lower limit, i.e. the smallest memory block that can be allocated.

This lower limit is necessary so that the overhead of storing used and free memory locations is minimized. If this lower limit did not exist, and many programs request small blocks of memory like 1K or 2K, the system would waste a lot of space trying to remember which blocks are allocated and unallocated. Typically this number would be a moderate number (like 2, so that memory is allocated in 2^2 = 4K blocks), small enough to minimize wasted space, but large enough to avoid excessive overhead. Let's call this lower limit *l*.

Now that we have our limits, let us see what happens when a program makes requests for memory. Let's say in this system, *l* = 6, which results in blocks 2^6 = 64K in size, and *u* = 10, which results in a largest possible allocatable block, 2^{10} = 1024K in size. The following shows a possible state of the system after various memory requests.

T	64 K	64 K	64 K	64 K	64 K	64 K	64 K	64 K	64 K	64 K	64 K	64 K	64 K	64 K	64 K	64 K
0	1024K															
1	A-64K	64K	128K		256K				512K							
2	A-64K	64K	B-128K		256K				512K							
3	A-64K	C-64K	B-128K		256K				512K							
4	A-64K	C-64K	B-128K	D-128K		128K			512K							
5	A-64K	64K	B-128K	D-128K		128K			512K							
6	128K		B-128K	D-128K		128K			512K							
7	256K			D-128K		128K			512K							
8	1024K															

Figure - 2

This allocation could have occurred in the following manner

1. Program A requests memory 34K..64K in size
2. Program B requests memory 66K..128K in size
3. Program C requests memory 35K..64K in size
4. Program D requests memory 67K..128K in size

5. Program C releases its memory
6. Program A releases its memory
7. Program B releases its memory
8. Program D releases its memory

As you can see, what happens when a memory request is made is as follows:

- If memory is to be allocated

1. Look for a memory slot of a suitable size (the minimal 2^k block that is larger or equal to that of the requested memory)
    1. If it is found, it is allocated to the program
    2. If not, it tries to make a suitable memory slot. The system does so by trying the following:
        1. Split a free memory slot larger than the requested memory size into half
        2. If the lower limit is reached, then allocate that amount of memory
        3. Go back to step 1 (look for a memory slot of a suitable size)
        4. Repeat this process until a suitable memory slot is found

- If memory is to be freed

1. Free the block of memory
2. Look at the neighboring block - is it free too?
3. If it is, combine the two, and go back to step 2 and repeat this process until either the upper limit is reached (all memory is freed), or until a non-free neighbour block is encountered

This method of freeing memory is rather efficient, as compaction is done relatively quickly, with the maximal number of compactions required equal to $\log_2(u/l)$ (i.e. $\log_2(u)- \log_2(l)$).

Typically the buddy memory allocation system is implemented with the use of a binary tree to represent used or unused split memory blocks.

However, there still exists the problem of internal fragmentation. In many situations, it is essential to minimize the amount of internal fragmentation.

6.2.1.4 Protection

Protection and sharing in systems with dynamic partitioning of memory are not significantly different from their counterparts in static partitioning environments, since they both rely on virtually identical hardware support. One difference is that dynamic partitioning potentially allows adjacent partitions in physical memory to overlap. Consequently, a single physical copy of a shared object may be accessible from two distinct address spaces. This possibility is illustrated in Figure 3, where partitions A and B overlap to include the shared object placed in the doubly shaded area. The relevant portion of the partition of the partition definition table is also shown in Figure 3. As indicated, 500 locations starting from the physical address 5500 are shared and included in both partitions. Although perhaps conceptually appealing, this form of sharing is quite restrictive in practice. Sharing of objects is limited to two processes; when several processes are in play, one of the more involved schemes described in above must be used.

6.2.1.5 Sharing

Sharing of code is generally more restrictive than sharing of data. One of the reasons for this is that shared code must be either reentrant or executed in a strictly mutually exclusive fashion with no preemption's. Otherwise, serious problems may result if a process in the middle of execution of the shared code is switched off, and another process begins to execute the same section of the shared code. Reentrancy generally requires that variables be kept on stack or in registers, so that new activation's do not affect the state of the preempted, incomplete executions. Additional complexities in sharing of code are imposed by the need for shared code to ensure that references to itself-such as local jumps and access to local data-are mapped properly during executions on behalf of any of the participating processes. When dynamic relocation with base registers is used, this means that all references to addresses within a shared-code object from instructions within that code must reach the same set of physical addresses where the shared code is stored at run-time, no matter which particular base is used for a given relocation. This may be accomplished in different ways, such as by making the shared code position independent or by having shared code

occupy identical virtual offsets in address spaces of all processes that reference it.

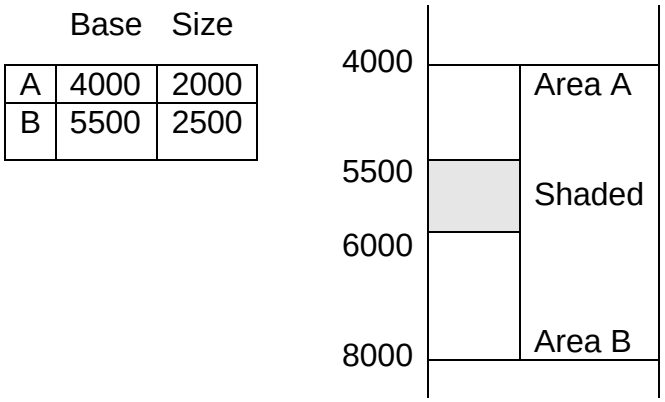


Figure 3 – Overlapping partitions

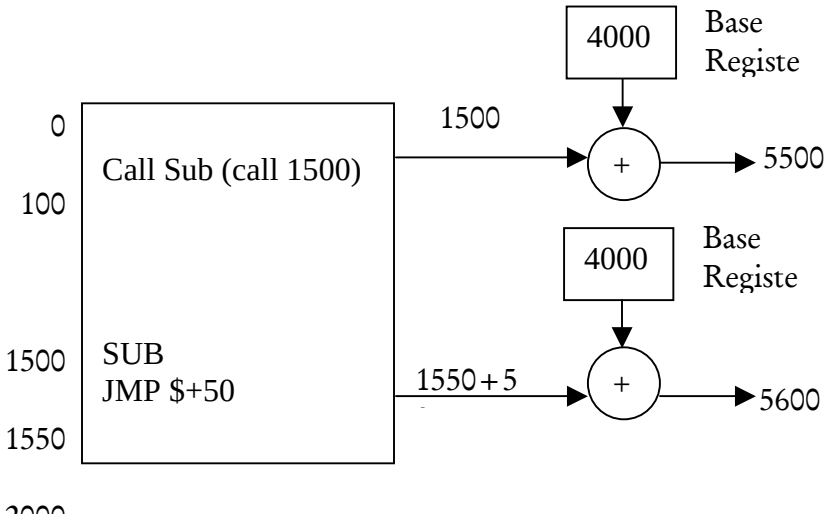
Some aspects of the issues involved in self-referencing of shared code are illustrated in Figure 4, where a subroutine SUB is assumed to be shared by two processes, PA and PB, whose respective partitions overlap in physical memory as indicated in Figure 3. Let us assume that the system in question used dynamic relocation and dynamic memory allocation, thus allowing the two partitions to overlap. The sizes of the address spaces of the two processes are 2000 and 2500 locations, respectively. The shared subroutine, SUB, occupies 500 locations, and it is placed in locations 5500 to 5999 in physical memory. The subroutine starts at virtual addresses 1500 and 0 in the address spaces of processes PA and PB, respectively. Being shared by the two processes, SUB may be linked with and loaded with either process image.

Figure 4 also shows the references to SUB from within the two processes. As indicated in Figure 4(a), the CALL SUB at virtual address 100 of process PA is mapped to the proper physical address of 5500 at run-time by adding the contents of PA's base register. Likewise the CASS SUB at virtual address 1800 in process PB is mapped to 5500 at run-time by adding PB's value of the base register. This is illustrated in Figure 4(b). Thus proper referencing of SUB from the two processes is accomplished even when the two partitions are relocated due to swapping or compaction, provided that they overlap in the same way in the new set of physical addresses. However, making references from within SUB

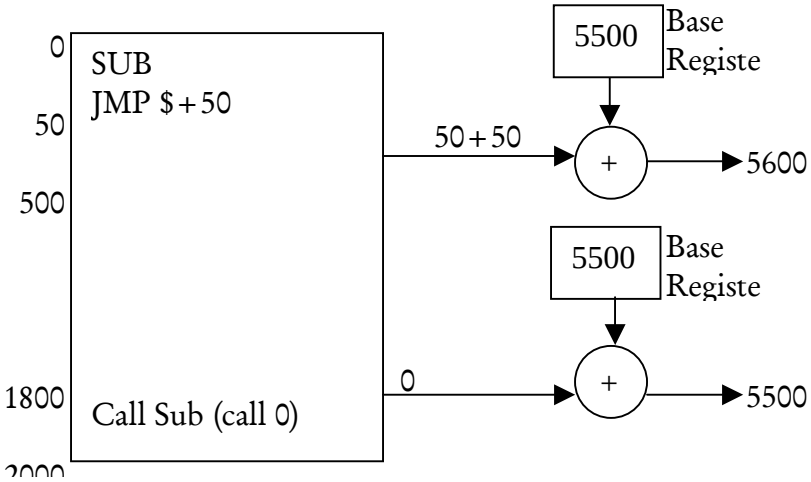
to itself poses a problem unless some special provisions are made. For example, a jump using absolute addressing from location 50 to location 100 within SUB should read JUMP 1600 for proper transfer of control when invoked by PA, but JMP 100 if PB's invocation is to work properly. Since the JMP instruction may have only one of these two addresses in its displacement field, there is a problem in executing SUB correctly in both possible contexts.

One way to solve this problem is to use relative references instead of absolute references within shared code. For example, the jump in question may read JMP $+50, where $ denotes the address of the JMP instruction. Since it is relative to the program counter, the JMP is mapped properly when invoked by either process, that is, to virtual address 1600 or 100, respectively. At run-time, however, both references map to the same physical address, 5600, as they should. This is illustrated in Figure 4.

Code that executes correctly regardless of its load address is often referred to as position-independent code. One of its properties is that references to portions of the position-independent code itself are always relative, say, to the program counter or to a base when based addressing is used. Position-independent coding is often used for shared code, such as memory-resident subroutine libraries. In our example, use of position-independent code solves the problem of self-referencing.



Process A



Process B

Figure 4 – Accessing shared code (a) Process A (b) Process B

Position-independent coding is one way to handle the problem of self-referencing of shared code. The main point of our example, however, is that sharing of code is more restrictive than sharing of data. In particular, both forms of sharing of code is more restrictive than sharing of data. In particular, both forms of sharing require the shared object to be accessible from all address spaces of which it is a part; in addition, shared code must also be reentrant or executed on a mutually exclusive basis, and some special provisions-such as position-independent coding-must be made in order to ensure proper code references to itself. Since ordinary (non-shared) code does not automatically meet these requirements, some special language provisions must be in place, or assembly language coding may be necessary to prepare shared code for execution in systems with partitioned allocation of memory.

6.2.1.6 Evaluation

- Wasted memory: This memory management scheme wastes less memory than fixed partitions because there is no internal fragmentation as the partition size can be of any length. By using compaction, external fragmentation can also be eliminated.
- Access Time: Access time is same as of fixed partitions as the same scheme of address translation using base register is used.

- Time Complexity: Time complexity is higher in variable partitions due to various data structures and algorithms used, for eg: Partition Description Table (PDT) is no more of fixed length.

6.3 Keywords

Variable partition: The number of partitions and their sizes are variable as they are not defined at the time of system generation.

Next fit: It 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 is the case with first fit.

Position-independent code: Code that executes correctly regardless of its load address is referred to as position-independent code.

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.

Compaction is to shuffle memory contents and place all free memory together in one block.

Program relocatability refers to the ability to load and execute a given program into an arbitrary place in memory.

6.4 SUMMARY

Fixed partition allocation although simple but has its own limitations in dealing with the problem of selection of a partition of suitable size. Variable (dynamic) partitioning allows allocation of the entire physical memory, except for the resident part of the Operating System, to a single process. Thus, in systems with dynamic partitioning, the virtual-address space of any given process or an overlay is limited only by the capacity of the physical memory in a given system. Dynamic creation of partitions according to the specific needs of requesting processes also eliminates the problem of internal fragmentation. Dynamic allocation of partitions requires the use of more complex algorithms for de-allocation of partitions and coalescing of free memory in order to combat external fragmentation. The fixed partition scheme suffers from internal fragmentation

while variable partitioning from external fragmentation. The need for occasional compaction of memory is also a major contributor to the increased time and space complexity of dynamic partitioning. Buddy system is a technique allocates memory in powers of 2, i.e 2^x, where x is an integer, thus facilitates memory compaction, although with a limitation of memory wastage.

Both fixed and variable partitioning of memory rely on virtually identical hardware support for relocation and protection. Sharing is quite restrictive in both systems.

6.5 SELF ASSESSMENT QUESTIONS (SAQ)

1. What do you understand by fragmentation? What is the difference between internal and external fragmentation? What is there in variable partitioned memory management? Explain. Use suitable example.
2. What do you understand by memory compaction? What are its merits and demerits? Discuss.
3. What are the different problems with memory compaction? Illustrate using suitable examples.
4. What is buddy system? How does it facilitate memory compaction? Discuss its advantages and disadvantages. Use suitable examples.
5. Discuss the merits and demerits of variable partitioned memory management scheme over fixed partitioned memory management scheme.
6. What do you understand by position independent coding? What are its advantages? Discuss.
7. Differentiate between First-fit and Next-fit allocation algorithms.
8. Compare the fixed partitioned memory management with variable partitioned memory management in terms of the problem of fragmentation.

6.6 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, Godbole A.S., 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.

Lesson number: 7 ***Writer: Dr. Rakesh Kumar***

Non-Contiguous Memory Management ***Vetter: Prof. Dharminder Kumar***

7.0 OBJECTIVE

The objective of this lesson is to introduce the concepts of Non-contiguous real memory management system. In the beginning Segmentation is also discussed which is an approach that contains the properties of contiguous as well as non-contiguous memory management systems.

7.1 INTRODUCTION

In Non-Contiguous Memory Management a program is divided into several blocks or segments that may be placed throughout main storage in pieces or chunks not necessarily adjacent to one another. It is the function of Operating System to manage these different chunks in such a way that they appear to be contiguous to the user.

At run time contiguous virtual address space is mapped to noncontiguous physical address space. This type of memory management is done in various ways:

1. Non-Contiguous, real memory management system
   - Paged memory management system
   - Segmented memory management system
   - Combined memory management system
2. Non-Contiguous, virtual memory management system
   - Virtual memory management system

7.2 PRESENTATION OF CONTENTS

7.2.1 Segmentation

7.2.1.1 Principles of Operation

7.2.1.2 Address Translation

7.2.1.3 Segment Descriptor Caching

7.2.1 SEGMENTATION

The external fragmentation and its negative impact should be reduced in systems where the average size of a request for allocation is smaller. Operating System cannot reduce the average process size, but a way to reduce the average size of a request for memory is to divide the address space of a single process into blocks that may be placed into noncontiguous areas of memory. This can be accomplished by segmentation. Segmentation provides breaking of an address space into several logical segments, dynamic relocation and sophisticated forms of protection and sharing.

According to user's view, programs are collections of subroutines, stacks, functions etc. Each of these components is of variable length and are logically related entities. Elements within segment are identified by their offset from beginning of the segment. Segments are formed at program translation time by grouping together logically related items. For example, a typical process may have separate code, data, and stack segments. Data or code shared with other processes may be placed in their own dedicated segments.

All segments of all programs do not have to be of the same length since the segments are formed as a result of logical division. 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. Since segments are not equal, segmentation is similar to dynamic partitioning. Thus segmentation possesses some properties of both contiguous (with regard to individual segments) and

noncontiguous (with regard to address space of a process) schemes for memory management.

Segmentation is quite natural for programmers who tend to think of their programs in terms of logically related entities, such as subroutines and global or local data areas. A segment is essential a collection of such entities. The segmented address space of a single process is illustrated in Figure 1 (a). In that particular example, four different segments are defined: DATA, STACK, CODE, and SHARED.

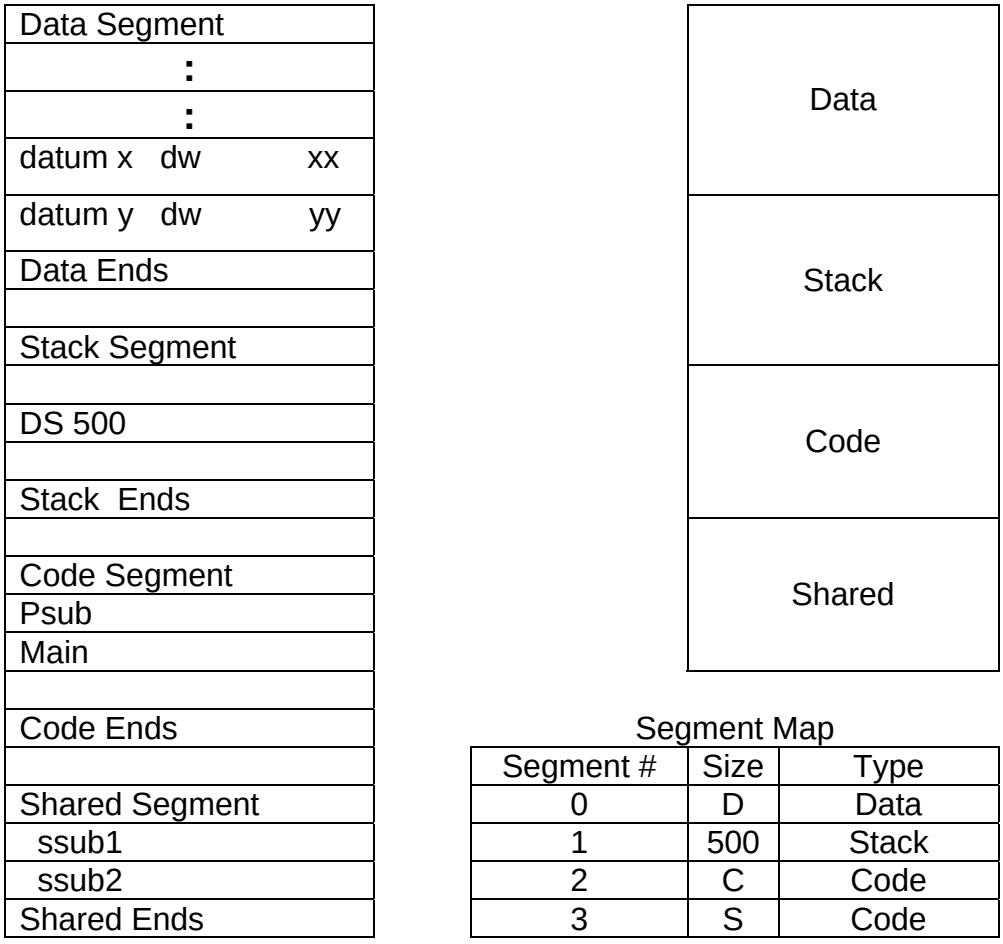


Figure 1 - Segments

Except for SHARED, the name of each segment is chosen to indicate the type of information that it contains. The STACK segment is assumed to consist of 500 locations reserved for stack. The SHARED segment consists of two subroutines, SSUB1 and SSUB2, shared with other processes. The definition of the segments follows the typical assembly-language notation, in which programmers usually

have the freedom to define segments directly in whatever way they feel best suits the needs of the program at hand. As a result, a specific process may have several different segments of the same generic type, such as code or data. For example, both CODE and SHARED segments contain executable instructions and thus belong to the generic type "code".

7.2.1.1 Principles of Operation

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>. In segmented systems, components belonging to a single segment reside in one contiguous area. But different segments belonging to same process occupy non-contiguous area of physical memory because each segment is individually relocated.

For example, the subroutine SSUB2 in segment SHARED is assumed to begin at offset 100. However, the unique designation of an item in a segmented address space requires the specification of both its segment and the relative offset therein. Offset 100 may fetch the first instruction of the subroutine SSUB2 within the segment SHARED, but the same relative offset may designate an entirely unrelated datum in the DATA segment.

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.

A sample linker-produced load module for the segments defined in Figure 1(a) is depicted in Figure 1(b). Virtual segment numbers are shown as part of the segment map that systems programs prepare to facilitate loading of segments in memory by the Operating System. When segment numbers and relative offsets within the segments are defined, two-component virtual addresses uniquely

identify all items within a process's address space. For example, if the SHARED segment is assigned number 3, the subroutine SSUB2 may be uniquely identified by its virtual address (3100), where 100 is the offset within the segment number 3-SHARED.

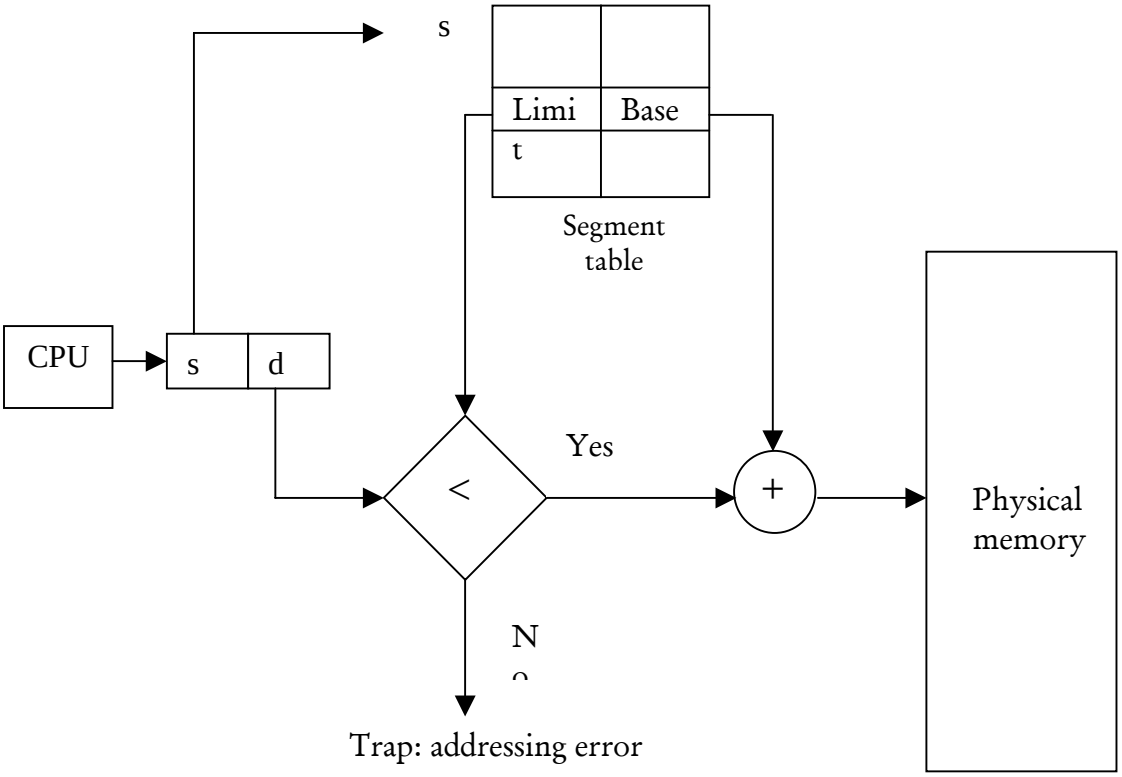
7.2.1.2 Address Translation

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 Operating System 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 2 illustrates a sample placement of the segments defined in Figure 1 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.



Hardware support for Segmentation

Figure 2 – Address translation in segmented systems

This process is illustrated in Figure 2 for the example of the virtual address (3100). To access Segment 3, the number 3 is used to index the SDT and to obtain the physical base address, 20000, of the segment SHARED. The size field of the same segment descriptor is used to check whether the supplied offset is within the legal bounds of its enclosing segment. If so, the base and offset are added to produce the target physical address. In our example that value is 20100, the first instruction word of the shared subroutine SSUB2.

In general, the size of a segment descriptor table is related to the size of the virtual address space of a process. For example, Intel's iAPX 286 processor is capable of supporting up to 16K segments of 64 KB each per process, thus

requiring 16K entries per SDT. Given their potential size, segment descriptor tables 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.

From the Operating System's point of view, segmentation is essentially a multiple-base-limit version of dynamically partitioned memory. Memory is allocated in the form of variable partitions; the main difference is that one such partition is allocated to each individual segment. Bases and limits of segments belonging to a given process are collected into an SDT are normally kept in the PCB of the owner process. Upon each process switch, the SDTBR and SDTLR are loaded with the base and size, respectively, of the SDT of the new running process. In addition to the process-loading time, SDT entries may also need to be updated whenever a process is swapped out or relocated for compaction purposes. Swapping out requires invalidation of all SDT entries that describe the affected segments. When the process is swapped back in, the base fields of its segment descriptors must be updated to reflect new load addresses. For this reason, swapping out of the SDT itself is rarely useful. Instead, the SDT of the swapped-out process may be discarded, and the static segment map-such as the one depicted in Figure 1(b)-may be used for creation of an up-to-date SDT whenever the related process is loaded in memory. Compaction, when supported, requires updating of the related SDT entry for each segment moved. In such systems, some additional or revised data structures may be needed to

facilitate identification of the SDT entry that describes the segment scheduled to be moved.

The price paid for segmenting the address space of a process is the overhead of storing and accessing segment descriptor tables. 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.

7.2.1.3 Segment Descriptor Caching

With performance of segmented systems so critically dependent on the duration of the address translation process, computer 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, the potential size of an SDT and the overhead of process switching make it too costly to keep an entire SDT of the running process in registers. A reasonable compromise is to keep a few of the most frequently used segment descriptors 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 Operating System'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.

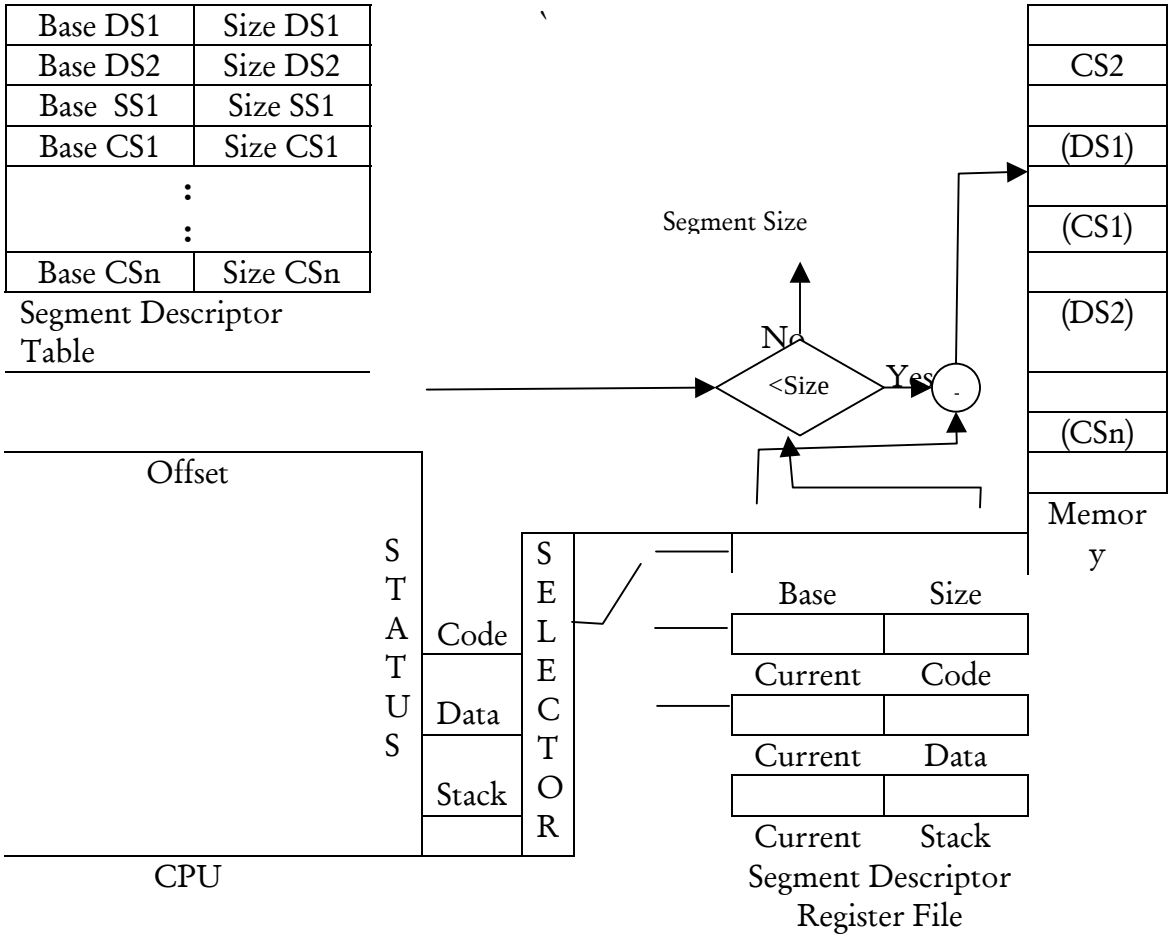


Figure 3 – Segment-descriptor cache registers

Register-assisted translation of virtual to physical addresses is illustrated in Figure 3. 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.

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 Operating System loads segment descriptor registers with their images recorded in the related PCB.

7.2.1.4 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. Thus most of the discussion of protection in systems with dynamic allocation of memory is applicable to segmented environments as well.

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, such as execution of data or modifications of code. 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 Operating System.

7.2.1.5 Sharing

Shared objects are usually placed in separate, dedicated segments. A shared segment may be mapped, via the appropriate segment descriptor tables, 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 4, where a code segment EMACS is assumed to be shared by three processes P1, P2 & P3. The relevant portions of the segment descriptor tables 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 4 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 4 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.

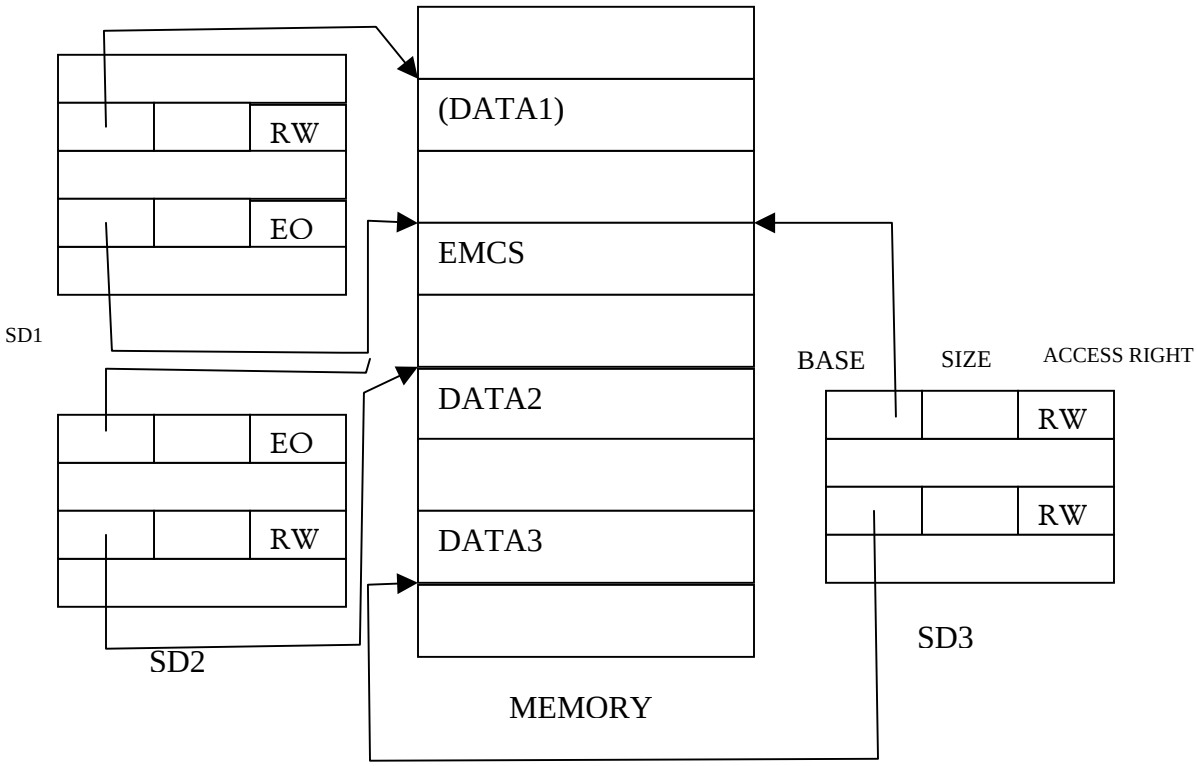


Figure 4 – Sharing in segmented systems

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.

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 Operating System must keep track of shared segments and of processes that access them. When a participating process is loaded in memory, the Operating System 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.

7.2.2 PAGING

Paging is another solution to the problem of memory fragmentation. It removes the requirement of contiguous allocation of physical memory. 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.

7.2.2.1 Principles of Operation

Figure 5 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 5. 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.

Address translation is performed with the help of the page-map table (PMT), constructed at process-loading time. As indicated in figure 5, 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 5. 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.

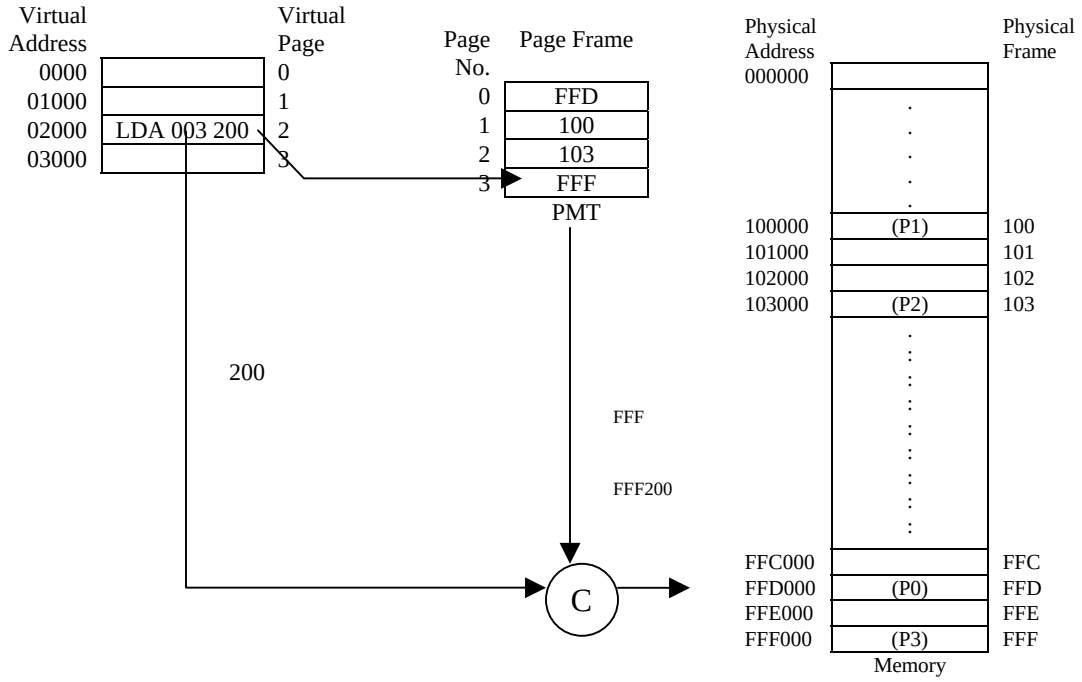


Figure 5 – Paging

The logic of the address translation process in paged systems is illustrated in Figure 5 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 Operating System keeps track of the status of each page frame by using a memory-map table (MMT). Format of an MMT is illustrated in Figure 6, assuming that only the process depicted in Figure 5 and the Operating System 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 6 – 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 Operating System must allocate n free page frames, so that ***n*** = Round(***s/p***) where ***p*** is the page size. The Operating System 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 Operating System 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.

7.2.2.2 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.

7.2.2.3 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 7.

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.

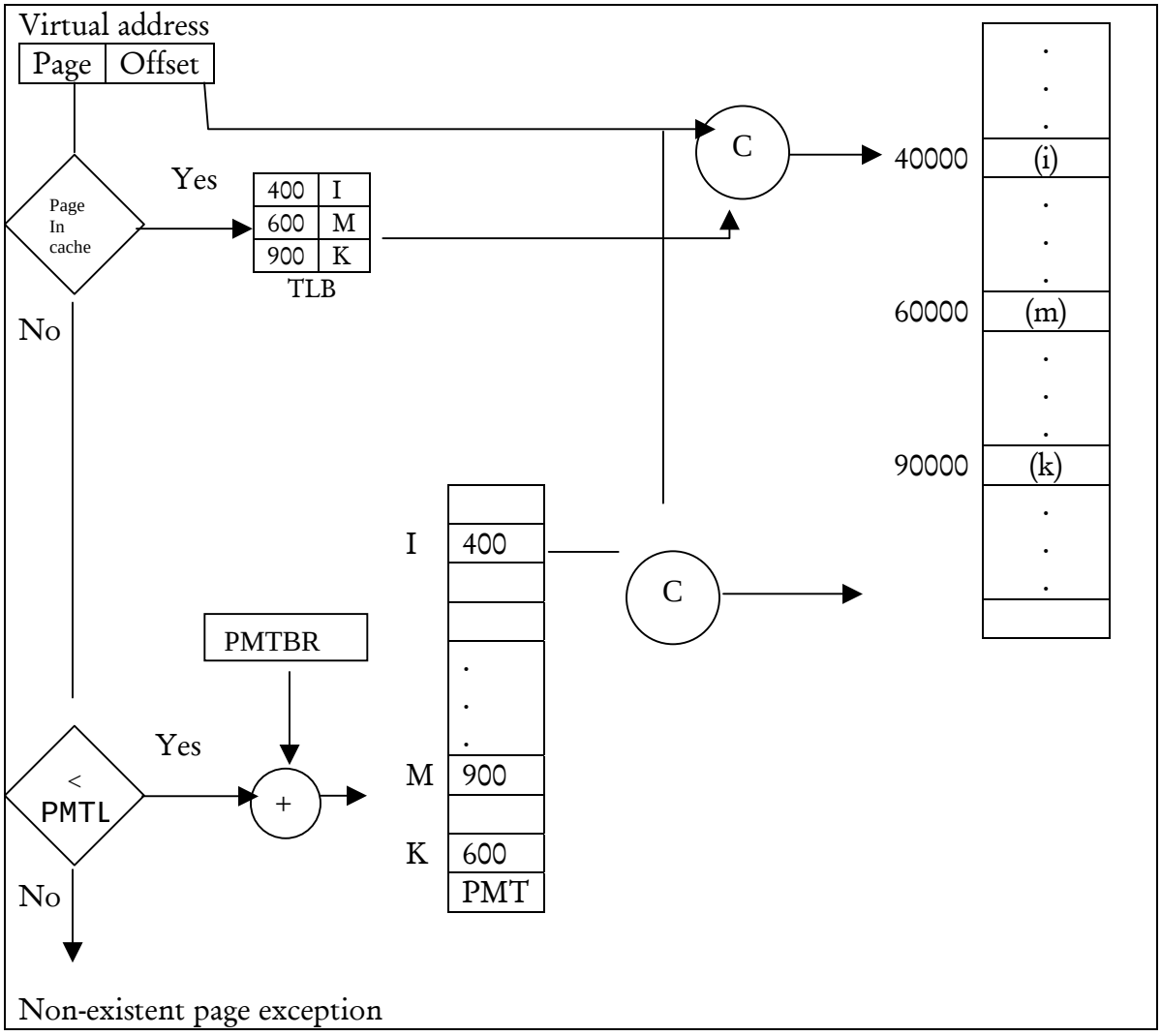


Figure 7 – Translation-lookaside buffer (TLB)

Figure 7 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.

$$t_{eff} = t_{TR} + t_M$$

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 $0 \le h \le 1$; 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.

7.2.2.4 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 Operating System 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.

7.3 Keywords

MMT: memory-map table (MMT) is used by the Operating System to keep track of the status of each page frame whether allocated or free.

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

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.

7.4 SUMMARY

Segmentation allows breaking of the virtual address space of a single process into separate entities (segments) 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. Unless virtual segmentation is supported, segmentation does not remove the problem of limiting the size of a process's virtual space by the size of the available physical memory.

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.

7.5 SELF ASSESSMENT QUESTIONS (SAQ)

1. What do you understand by segmentation? Discuss in detail the address translation mechanism in segmentation.
2. Write a detailed mote on sharing in segmentation. Also discuss the problem during swapping in it.
3. How the access rights are implementation in sharing in segmentation.
4. What is the basic difference between paging and segmentation? Which one is better and why?
5. What is the difference between a segment and a page? Discuss using suitable example.
6. What is Table Look aside Buffer (TLB)? How is it used to speed up the memory access? Explain.

7.6 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, Godbole A.S., 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.

Lesson number: 8	***Writer: Dr. Rakesh Kumar***
Virtual Memory	***Vetter: Prof. Dharminder Kumar***

8.0 OBJECTIVE

This lesson is focused on the non-contiguous memory management systems. The objective of this lesson is to make the students primarily familiar with the following concepts:

(a) Virtual memory.

(b) Page replacement policies

(c) Page allocation policies

(d) Segmentation and Paging

8.1 INTRODUCTION

Virtual memory is designed to solve the problem of running a program that needs more memory than the hardware has. One way of approaching this is to use overlays. Overlays are code and data written to memory under system or programmer control to reuse memory for a process. The old memory could be overwritten or saved first to disk for later use as another overlay. Programmers had to create the overlays, which required laying out their code in such a way that it could be overlaid. Virtual Memory provides the same functionality, and solved the protection and relocation problems in an interesting way. Central to Virtual Memory is the idea of a virtual address and the associated virtual address space. 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. The open issues are how the virtual to physical translation is made, and how this all allows automatic overlays.

8.2 Presentation of contents

8.2.1 Virtual Memory

- 8.2.1.1 Principles of Operation
- 8.2.1.2 Management of Virtual Memory
- 8.2.1.3 Program Behavior
- 8.2.1.4 Replacement Policies
  - 8.2.1.4.1 Replacement Algorithms
  - 8.2.1.4.2 Global and Local Replacement Policies
- 8.2.1.5 Allocation Policies
- 8.2.1.6 Hardware Support and Considerations
- 8.2.1.7 Protection and Sharing

8.2.2 Segmentation and Paging

8.2.1 VIRTUAL 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 Operating System 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. Virtual memory provides the illusion of a much larger memory than may actually be available, so programmers are relieved of the burden of trying to fit a program into limited memory.

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 Operating System 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.

8.2.1.1 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 (Page Map Table), SDT (Segment Descriptor Tables), or both. The important characteristic is that in virtual-memory systems some portions of the address space of the running process can be absent from main memory.

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 Operating System 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. In other words, 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 page-map tables. 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 Operating System clears all the presence bits in the related page-map table. As and when specific pages are brought into the main memory, its presence bit is reset.

A possible implementation is illustrated in Figure 1. 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.

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. In paged virtual-memory systems, this exception is called a 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 Operating System 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 1.

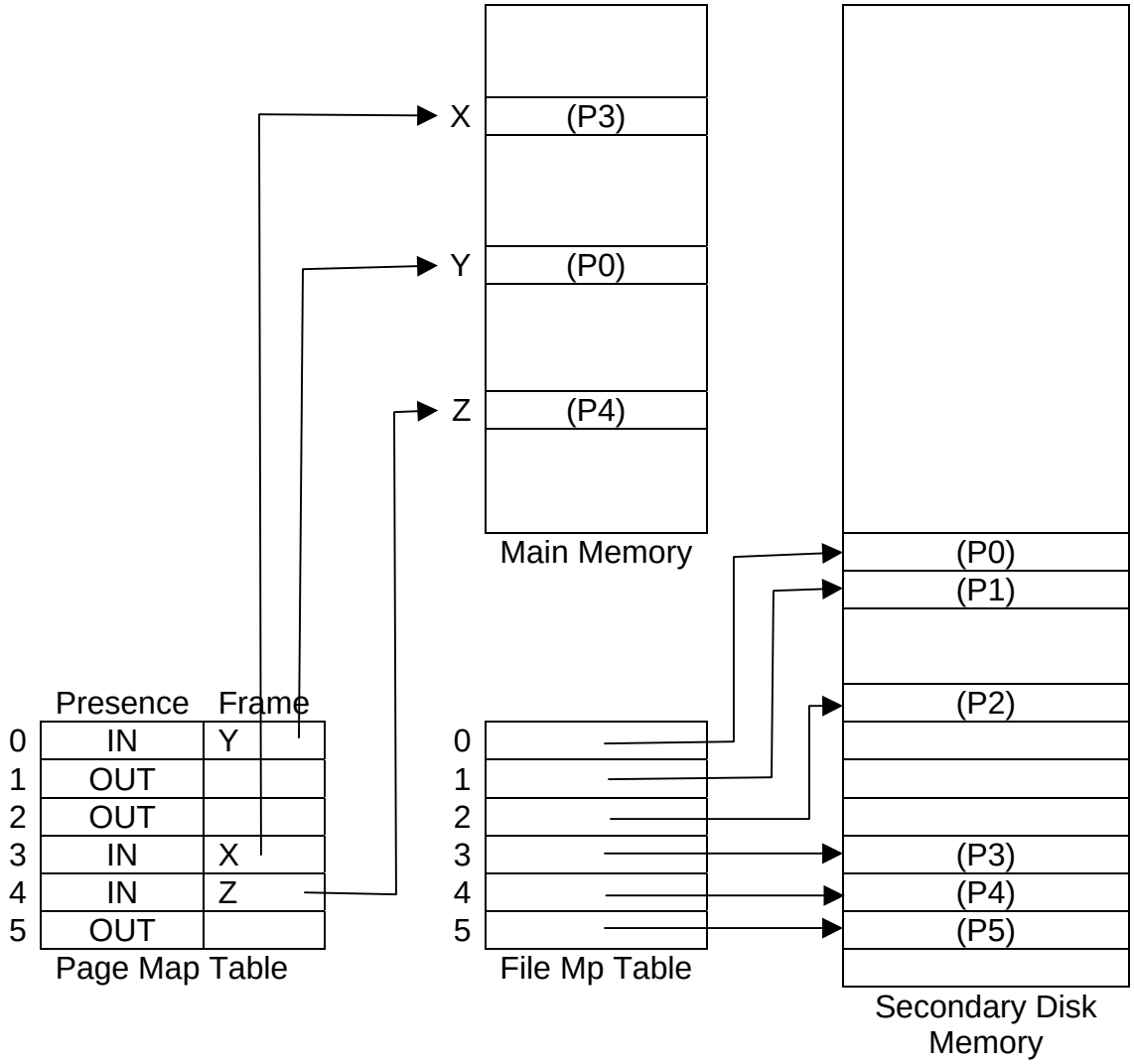


Figure 1: Virtual Memory

Page Faults

When a page is referenced and not found in the main memory, the Operating System faces a page fault. The following are the basic steps in servicing a page fault:

- The MMU interrupts the CPU (with a paging interrupt or exception depending on the CPU model)

- The paging ISR i.e. Interrupt Service Routine (usually called a pager or page fault handler) loads the required page into an available frame. If a frame is not available, the pager must make one available, by discarding a frame. It's generally better to evict a clean page than a dirty one because dirty ones must be written to backing store. The current process may give up the CPU while the relevant pages are moved in and out. (This is basically an I/O request, although not an explicit one).
- When the pages are available and the process is runnable again, the faulting instruction is restarted.

8.2.1.2 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 Operating System 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

8.2.1.3 Program Behavior

Program behavior is of extreme importance for the performance of virtual-memory systems. Execution of partially loaded programs generally leads to longer turnaround times due to the processing of page faults. 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 is the tendency for a program to reference clustered locations in preference to randomly distributed locations. 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.

A locality is a small cluster of not necessarily adjacent pages to which most memory references are made during a period of time. 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.

8.2.1.4 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.

8.2.1.4.1 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.

Belady's Anomaly

A group of researchers, led by a fellow named Belady, discovered a surprising fact about FIFO paging. It's possible (though unlikely) that adding memory to a FIFO paging system increases the number of faults. The example is below:

Reference Strings	0	1	2	3	0	1	4	0	1	2	3	4
Youngest Page	0	1	2	3	0	1	4	4	4	2	3	3
		0	1	2	3	0	1	1	1	4	2	2
Older Page			0	1	2	3	0	0	0	1	4	4
Page Faults	P	P	P	P	P	P	P			P	P	

Figure 2(a): Number of real pages 3

Reference Strings	0	1	2	3	0	1	4	0	1	2	3	4
Youngest Page	0	1	2	3	3	3	4	0	1	2	3	4
		0	1	2	2	2	3	4	0	1	2	3
			0	1	1	1	2	3	4	0	1	2
Older Page				0	0	0	1	2	3	4	0	1
Page Faults	P	P	P	P			P	P	P	P	P	P

Figure 2(b): Number of real pages 4

This result has been generalized, and the key property is called the stack property: that increasing the size of memory only adds contents to it. In the FIFO case above, there are different contents in the upper 3 page frames in memory for several states. FIFO is not a stack algorithm. Any non-stack algorithm can display Belady's anomaly. LRU is a stack algorithm.

Least Recently Used (LRU):

The least recently used 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 and 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.

8.2.1.4.2 Global and Local Replacement Policies

As discussed, all replacement policies choose a victim among the resident pages owned by the process that experiences the page fault. This is known as local replacement. However, each of the presented algorithms may be made to operate globally. A global replacement algorithm processes all resident pages when selecting a victim. Local replacement tends to localize effects of the

allocation policy to each particular process. Global replacement, on the other hand, increases the degree of coupling between replacement and allocation strategies. A global replacement algorithm may take pages allocated to one process by the allocation algorithm, away. Global replacement is concerned mostly with the overall state of the system, and much less with the behavior of each individual process. Global replacement is known to be sub-optimal.

8.2.1.5 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.

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 Operating System 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 Operating System 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 Operating System defines a system-wide (or per-process) critical page-fault frequency, P.
2. The Operating System 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 Operating System 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 Operating System 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.

8.2.1.6 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 is an important design consideration in that it 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 necessary to position the heads over the target track and sector, and the page-transfer time necessary to transfer the page to main memory thereafter. 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. Research results suggest that procedures in many applications tend to be smaller than 100 words. 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, larger pages 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. In general, the page-size trade-off is technology-dependent, and its outcome tends to vary as the price and performance of individual components change.

8.2.1.7 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 or 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 Operating System itself as a shared object. As such, the Operating System 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 Operating System. 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 Operating System. Moreover, the Operating System-calling mechanism may be simplified by avoiding expensive mode switches between users and the Operating System code. With the protection mechanism provided by mapping, a much faster CALL instruction, or its variant, may be used to invoke the Operating System.

8.2.2 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 Operating System 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 5. 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.

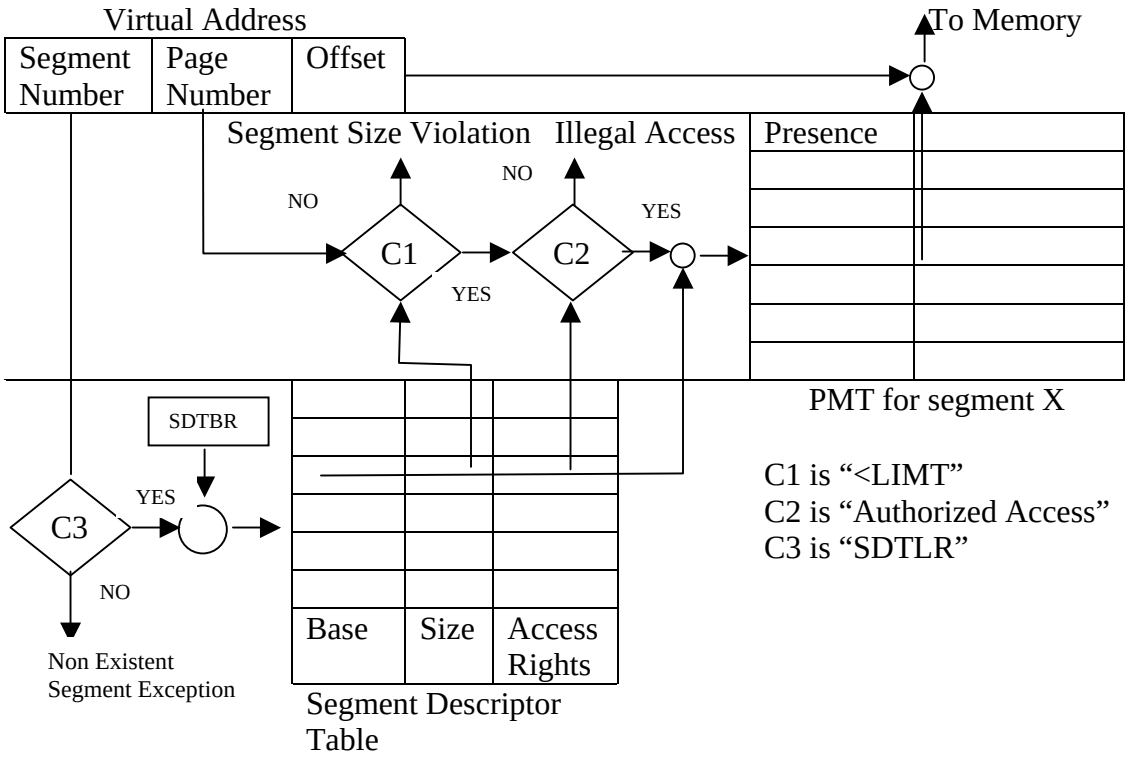


Figure 5 – Segmentation and paging

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.

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. It may be too much to bear even in the face of all the added benefits. Obviously, hardware designers of such systems must assist the work of the Operating System by providing ample support in terms of mapping registers and look aside buffers.

8.3 Keywords

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.

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

Dirty page: A page on which a write operation has been performed.

Clean page: A page which is not dirty i.e. not modified due to write operation.

Thrashing: A process is thrashing if it spending more time in paging (i.e. page swapping) then executing.

8.4 SUMMARY

The memory-management layer of an Operating System allocates and reclaims portions of main memory in response to requests, and in accordance with the resource-management objectives of a particular system. Memory is normally freed when resident objects terminate. When it is necessary and cost-effective, the memory manager may increase the amount of available memory by moving inactive or low-priority objects to lower levels of the memory hierarchy (swapping). The objective of memory management is to provide efficient use of memory by minimizing the amount of wasted memory while imposing little storage, computational, and memory-access overhead. In addition, the memory

manager should provide protection by isolating distinct address spaces, and facilitate inter-process cooperation by allowing access to shared data and code.

It is very desirable to execute a process whose logical address space is larger than the available physical address space. It can be achieved through overlays but imposing a lot of burden on the programmers. The better 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.

8.6 SELF ASSESMENT QUESTIONS (SAQ)

1. Write short notes on following:
   (a) Thrashing
   (b) Page fault frequency
   (c) Sharing in virtual memory

2. 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
3. What is the common drawback of all the real memory management techniques? How is it overcome in virtual memory management schemes?
4. What extra hardware do we require for implementing demand paging and demand segmentation?
5. Show that LRU page replacement policy possesses the stack property.
6. Differentiate between internal and external fragmentation.
7. What do you understand by thrashing? What are the factors causing it?
8. Compare FIFO page replacement policy with LRU page replacement on the basis of overhead.
9. What do you understand by Belady's anomaly? Show that page replacement algorithm which possesses the stack property cannot suffer from Belady's anomaly.

8.5 SUGGESTED READINGS / REFERENCE MATERIAL

1. Operating Systems 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. Operting 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.