Module I
OPERATING SYSTEM

An operating system (OS) is the program that, after being initially loaded into the computer by a boot program, manages all the other programs in a computer. The other programs are called applications or application programs. The application programs make use of the operating system by making requests for services through a defined application program interface (API). In addition, users can interact directly with the operating system through a user interface such as a command line or a graphical user interface (GUI).

An Operating System (OS) is an interface between a computer user and computer hardware. An operating system is a software which performs all the basic tasks like file management, memory management, process management, handling input and output, and controlling peripheral devices such as disk drives and printers. Some popular Operating Systems include Linux, Windows, UNIX, MS-DOS, MS-Windows - 98/XP/Vista, Windows-NT/2000, OS/2 and Mac OS, OS X, VMS, OS/400, AIX, z/OS, etc.

An operating system (OS) is system software that manages computer hardware and software resources and provides common services for computer programs. All computer programs, excluding firmware, require an operating system to function.

The operating system (OS) is the most important program that runs on a computer. Every general-purpose computer must have an operating system to run other programs and applications. Computer operating systems perform basic tasks, such as recognizing input from the keyboard, sending output to the display screen, keeping track of files and directories on the storage drives, and controlling peripheral devices, such as printers.

For large systems, the operating system has even greater responsibilities and powers. It is like a traffic cop, it makes sure that different programs and users running at the same time do not interfere with each other. The operating system is also responsible for security, ensuring that unauthorized users do not access the system.

An operating system, or "OS," is software that communicates with the hardware and allows other programs to run. It is comprised of system software, or the fundamental files our computer needs to boot up and function. Every desktop computer, tablet, and Smartphone includes an operating system that provides basic functionality for the device.

Operating System is software that works as an interface between a user and the computer hardware. The primary objective of an operating system is to make computer system convenient to use and to utilize computer hardware in an efficient manner. The operating system performs the basic tasks such as receiving input from the keyboard, processing instructions and sending output to the screen. Operating system is software that is required in order to run application programs and utilities. It works as a bridge to perform better interaction between application programs and hardware of the computer.

Objectives of Operating Systems

- **Convenience**: An OS makes a computer more convenient to use.
- **Efficiency**: An OS allows the computer system resources to be used in an efficient manner.
- **Ability to evolve**: An OS should be constructed in such a way as to permit the effective development, testing, and introduction of new system functions without interfering with service.

The other objectives are:

- To make the computer system convenient to use in an efficient manner.
- To hide the details of the hardware resources from the users.
- To provide users a convenient interface to use the computer system.
- To act as an intermediary between the hardware and its users, making it easier for the users to access and use other resources.
- To manage the resources of a computer system.
- To keep track of who is using which resource, granting resource requests, and mediating conflicting requests from different programs and users.

- To provide efficient and fair sharing of resources among users and programs.

OS as resource manager – Efficiency

It is not the OS itself but the hardware that makes all kinds of services possible and available to application programs. An OS merely exploits the hardware to provide easily accessible interfaces. Exploitation means management upon the hardware resources, and thus also imposes control upon or manages the entities that use the services so that the resources are used efficiently. In the classes later on, we will discuss this aspect, including process scheduling, memory management, I/O device management, etc. One thing worth mentioning here is that, different from other control systems where the controlling facility, the controller, is distinct and external to the controlled parts, the OS has to depend on the hardware resources it manages to work.

As we know, an OS is in nature a program, consisting instructions, thus it also needs CPU to execute instructions so as to function as a controller, and main memory to hold instructions for CPU to fetch. At the same time, the OS has to be able to relinquish and regain later the control of CPU so that other programs can get chance to run but still under the control of the OS. By utilizing the facilities provided by hardware, the OS may schedule different processes to run at different moments and exchange the instructions and data of programs between external storage devices, like hard disks, and main memory. These topics will be covered as the course proceeds.

Functions of Operating Systems

Following are some of important functions of an operating System.

1. Memory Management
2. Processor Management
3. Device Management
4. File Management
5. Security
6. Control over system performance
7. Job accounting
8. Error detecting aids
9. Coordination between other software and users

Memory Management

Memory management refers to management of Primary Memory or Main Memory. Main memory is a large array of words or bytes where each word or byte has its own address. Main memory provides a fast storage that can be accessed directly by the CPU. For a program to be executed, it must in the main memory. An Operating System does the following activities for memory management.

- Keeps tracks of primary memory, i.e., what part of it are in use by whom, what part is not in use?
- In multiprogramming, the OS decides which process will get memory when and how much.
- Allocates the memory when a process requests it to do so.
- De-allocates the memory when a process no longer needs it or has been terminated.

Processor Management

In multiprogramming environment, the OS decides which process gets the processor when and for how much time. This function is called process scheduling. An Operating System does the following activities for processor management.

- Keeps tracks of processor and status of process. The program responsible for this task is known as traffic controller.
- Allocates the processor (CPU) to a process.
- De-allocates processor when a process is no longer required.

Device Management

An Operating System manages device communication via their respective drivers. It does the following activities for device management

- Keeps tracks of all devices. Program responsible for this task is known as the I/O controller.
- Decides which process gets the device when and for how much time.
- Allocates the device in the efficient way.
- De-allocates devices.

File Management

A file system is normally organized into directories for easy navigation and usage. These directories may contain files and other directions.

An Operating System does the following activities for file management.

- Keeps track of information, location, uses, status etc. The collective facilities are often known as file system.
- Decides who gets the resources.
- Allocates the resources.
- De-allocates the resources.

Other Important Activities

- **Security** − By means of password and similar other techniques, it prevents unauthorized access to programs and data.
- **Control over system performance** − Recording delays between request for a service and response from the system.
- **Job accounting** − Keeping track of time and resources used by various jobs and users.
- **Error detecting aids** − Production of dumps, traces, error messages, and other debugging and error detecting aids.
- **Coordination between other softwares and users** − Coordination and assignment of compilers, interpreters, assemblers and other software to the various users of the computer systems.

The Evolution of Operating Systems

Point out that the evolution of operating system software parallels the evolution of the computer hardware they were designed to control.

1940s

1. Provide students with an overview of first-generation computers, which were based on vacuum tube technology. Point out that there was no standard operating system software at that time.

2. Mention that a typical program included every instruction needed by the computer to perform the tasks requested, and the machines were poorly utilized, i.e., the CPU processed data and made calculations for only a fraction of the available time. Basically, early programs were designed to

use the resources conservatively at the expense of understand ability.

1945: ENIAC, Moore School of Engineering, University of Pennsylvania.
1949: EDSAC and EDVAC
1949 BINAC - a successor to the ENIAC

1950s

1. Provide students with an overview of second-generation computers (1955-1965). Point out that business environments placed importance on cost effectiveness; however, computers were still very expensive.

2. Outline two major improvements that were widely adopted: computer operators were hired to facilitate each machine's operation, and job scheduling was instituted.

3. Use examples to show how job scheduling helps improve productivity. Point out that job scheduling introduced the need for control cards, which defined the exact nature of each program and its requirements. Discuss job control language (JCL).

4. Discuss various factors that helped improve the performance of the CPU, such as an increase in the speed of I/O devices, the development of access methods, the introduction of buffers and timer interrupts, etc.

5. Point out that during this time, techniques were developed to manage program libraries, create and maintain data files and indexes, randomize direct access addresses, and create and check file labels. However, programs were still run in serial batch mode, one at a time.

1951: UNIVAC by Remington
1952: IBM 701
1956: The interrupt
1954-1957: FORTRAN was developed

1960s

1. Provide students with an overview of third-generation computers dated from the mid-1960s. Point out that they were designed with faster CPUs, but their speed caused problems when they interacted with the relatively slow I/O devices.

2. Explain how the concept of multiprogramming helped solve this problem. Discuss the mechanism of its implementation.

3. Use examples to explain the concepts of passive multiprogramming and active multiprogramming. Point out the disadvantages of passive multiprogramming and how these were overcome by active multiprogramming.

4. Point out that in this generation, few advances were made in data management, and the total operating system was customized to suit users' needs.

1960s: Disks become mainstream
1961: The dawn of minicomputers
1962 Compatible Time-Sharing System (CTSS) from MIT
1963 Burroughs Master Control Program (MCP) for the B5000 system
1964: IBM System/360
1966: Minicomputers get cheaper, more powerful, and really useful
1967-1968: The mouse
1964 and onward: Multics
1969: The UNIX Time-Sharing System from Bell Telephone Laboratories

1970s

1. Point out that during the late 1970s, computers had faster CPUs, thus creating an even greater disparity between their rapid processing speed and slower I/O access time. Multiprogramming schemes to increase CPU use were limited by physical capacity of main memory.
2. Discuss how the concept of virtual memory solved the physical limitation issue.

3. Point out some other important developments during this time: database management software became a popular tool; a number of query systems were introduced; and programs started using English-like words, modular structures, and standard operations.

4. Discuss the implications this supercomputer had on computer systems and operating system design.

- Multi User and Multi tasking was introduced.
- Dynamic address translation hardware and Virtual machines came into picture.
- Modular architectures came into existence.
- Personal, interactive systems came into existence.

1971: Intel announces the microprocessor
1972: IBM comes out with VM: the Virtual Machine Operating System
1973: UNIX 4th Edition is published
1973: Ethernet
1974 The Personal Computer Age begins
1974: Gates and Allen wrote BASIC for the Altair
1976: Apple II

1980s

1. Discuss the various developments in the 1980s, such as improved cost/performance ratio of computer components, greater flexibility of hardware, and the introduction of the concept of firmware, etc.

2. Provide students with an overview of multiprocessing, which was introduced during this time and allowed the parallel execution of programs.

3. Point out that the evolution of personal computers and high-speed communications sparked the move to distributed processing and networked systems, enabling users in remote locations to share hardware and software resources.

4. Provide students with an overview of network operating systems and distributed operating systems.

- August 12, 1981: IBM introduces the IBM PC
- 1983 Microsoft begins work on MS-Windows
- 1984 Apple Macintosh comes out

1990s

1. Point out that the demand for Internet capability in the mid-1990s sparked the proliferation of networking capability. The World Wide Web, conceived by Tim Berners-Lee, made the Internet accessible by computer users worldwide.

2. Be sure to note that increased networking also created increased demand for tighter security to protect hardware and software.

3. Point out that the decade also introduced a proliferation of multimedia applications demanding additional power, flexibility, and device compatibility for most operating systems.

4. Tim Berners-Lee, the person who is credited with designing the first World Wide Web server. Discuss the implications of this design on today's computing environments.

1990 Microsoft Windows 3.0 comes out
1991 GNU/Linux
1992 The first Windows virus comes out
1993 Windows NT

2000s

1. Point out that primary design features of current operating systems are based on providing support for various services such as multimedia, Internet and Web access, and client/server computing, etc. Outline various requirements of computer systems in order to meet these demands, such as increased CPU speed, high-speed network attachments, and increased number and variety of storage devices.
2. Explain the process of virtualization.
3. Discuss recent advances in processing speeds.

2007: iOS
2008: Android OS

Serial Processing:

- Early computer from late 1940 to the mid 1950.
- The programmer interacted directly with the computer hardware.
- These machine are called bare machine as they don't have OS.
- Every computer system is programmed in its machine language.
- Uses Punch Card, paper tapes and language translator

These system presented two major problems.
1. Scheduling
2. Set up time

Scheduling: Used signup sheet to reserve machine time. A user may sign up for an hour but finishes his job in 45 minutes. This would result in wasted computer idle time, also the user might run into the problem not finish his job in allotted time.

Set up time: A single program involves:

- Loading compiler and source program in memory
- Saving the compiled program (object code)
- Loading and linking together object program and common function

Each of these steps involves the mounting or dismounting tapes on setting up punch cards. If an error occur user had to go the beginning of the set up sequence. Thus, a considerable amount of time is spent in setting up the program to run. This mode of operation is turned as serial processing ,reflecting the fact that users access the computer in series. Simple

Simple batch Systems

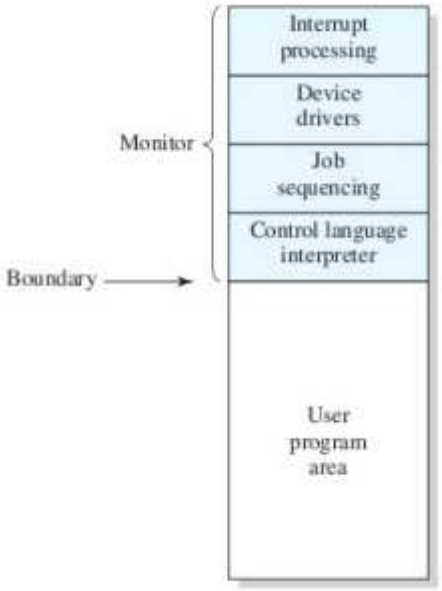
The users of a batch operating system do not interact with the computer directly. Each user prepares his job on an off-line device like punch cards and submits it to the computer operator. To speed up processing, jobs with similar needs are batched together and run as a group. The programmers leave their programs with the operator and the operator then sorts the programs with similar requirements into batches.

The problems with Batch Systems are as follows

- Lack of interaction between the user and the job.
- CPU is often idle, because the speed of the mechanical I/O devices is slower than the CPU.
- Difficult to provide the desired priority.
- Early computers were very expensive, and therefore it was important to maximize processor utilization.
- The wasted time due to scheduling and setup time in Serial Processing was unacceptable.
- To improve utilization, the concept of a batch operating system was developed.

Batch is defined as a group of jobs with similar needs. The operating system allows users to form batches. Computer executes each batch sequentially, processing all jobs of a batch considering them as a single process called batch processing.

The central idea behind the simple batch-processing scheme is the use of a piece of software known as the monitor. With this type of OS, the user no longer has direct access to the processor. Instead, the user submits the job on cards or tape to a computer operator, who batches the jobs together sequentially and places the entire batch on an input device, for use by the monitor. Each program is constructed to branch back to the monitor when it completes processing, at which point the monitor automatically begins loading the next program.



Memory Layout for resident memory

With a batch operating system, processor time alternates between execution of user programs and execution of the monitor. There have been two sacrifices: Some main memory is now given over to the monitor and some processor time is consumed by the monitor. Both of these are forms of overhead.

Multi Programmed batch Systems

A single program cannot keep either CPU or I/O devices busy at all times. Multiprogramming increases CPU utilization by organizing jobs in such a manner that CPU has always one job to execute. If computer is required to run several programs at the same time, the processor could be kept busy for the most of the time by switching its attention from one program to the next. Additionally I/O transfer could overlap the processor activity i.e, while one program is waiting for an I/O transfer; another program can use the processor. So CPU never sits idle or if comes in idle state then after a very small time it is again busy.

In multi-programmed batched operating systems, the operating system reads jobs from disk drives where a list of jobs is already being stored through card readers. The operating system then pulls and stores as much job as it can in the memory. Then from the memory, operating system start working on a job. Now, whenever a job reaches a situation where is has to be waiting for one or more tasks to be completed like use of any IO devices, the operating system pulls another job from the memory and starts working on it. Whenever this job also starts waiting, for example it need to use the same IO which is already in use by its previous job, the operating systems pulls another job. This is how, a multi-programmed batched systems harness the power of disk drives and memory.

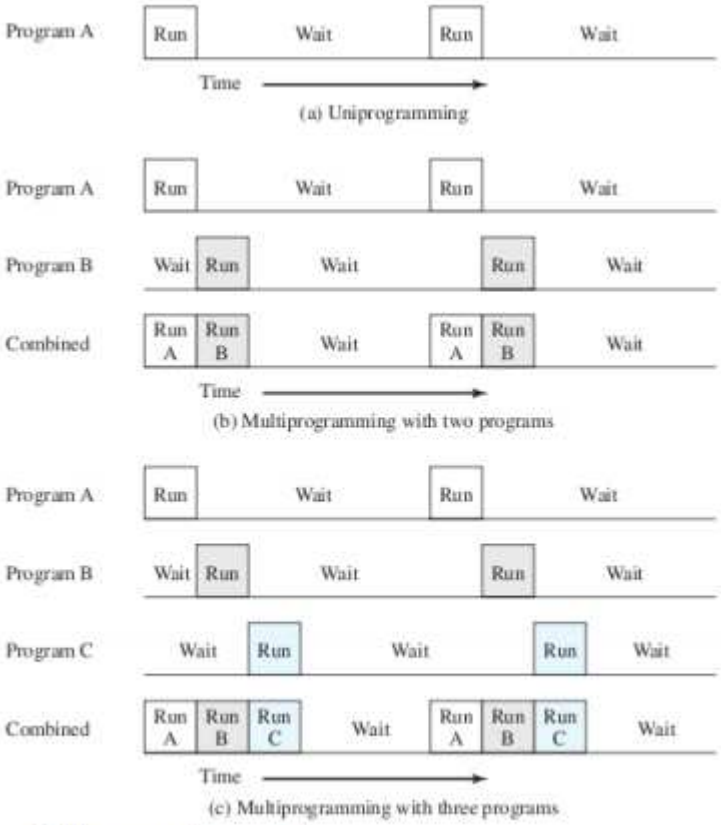
In multiprogramming, multiple programs (or jobs) of different users can be executed simultaneously (i.e. at the same time). The multiple jobs that have to be run simultaneously must be kept in main memory and the operating system must manage them properly. If these jobs are ready to run, the processor must decide which one to run.

In multi-programmed batch system, the operating system keeps multiple jobs in main memory at a time. There may be many jobs that enter the system. Since in general, the main memory is too small to accommodate all jobs. So the jobs that enter the system to be executed are kept initially on the disk in the job pool. In other words, we can say that a job pool consists of all jobs residing on the disk awaiting allocation of main memory. When the operating system selects a job from a job pool, it loads that job into memory for execution.

Normally, the jobs in main memory are smaller than the jobs in job pool. The jobs in job pool are awaiting allocation of main memory. If several jobs are ready to be brought into memory, and if there is not enough room for all of them, then the system must require memory management. Similarly, if many jobs are ready to run at the same time, the system must schedule these jobs.

The processor picks and begins to execute one of the jobs in main memory. Some jobs may have to wait for certain tasks (such as I/O operation), to complete. In a simple batch system or non-multi-programmed system, the processor would sit idle. In multi-programmed system, the CPU switches to second job and begins to execute it. Similarly, when second job needs to wait, the processor is switched to third job, and so on. The processor also checks the status of previous jobs, whether they are completed or not.

The multi-programmed system takes less time to complete the same jobs than the simple batch system. The multi-programmed systems do not allow interaction between the processes (or jobs) when they are running on the computer. Multiprogramming increases the CPU's utilization. Multi-programmed system provides an environment in which various computer resources are utilized effectively. The CPU always remains busy to run one of the jobs until all jobs complete their execution. In multi-programmed system, the hardware must have the facilities to support multiprogramming.



Multiprogramming example

Sl.No.	Simple Batched Systems	Multi-programmed Batched Systems
1	In this system, processes are processed one after another	In this system, multiple processes can be executed at a time.
2	As one process gets processed at a time, it performs low.	Processes are executed in a parallel fashion, thus it is faster.
3	CPU remains in idle states for long times.	CPU do not need to remain in idle state.
4	Example: CP/M, MS DOS, PC DOS etc.	Example: Windows 95, MaxOS etc.

Time Sharing Systems

Time-sharing is a technique which enables many people, located at various terminals, to use a particular computer system at the same time. Time-sharing or multitasking is a logical extension of multiprogramming. Processor's time which is shared among multiple users simultaneously is termed as time-sharing.

The main difference between Multiprogrammed Batch Systems and Time-Sharing Systems is that in case of Multiprogrammed batch systems, the objective is to maximize processor use, whereas in Time-Sharing Systems, the objective is to minimize response time.

Multiple jobs are executed by the CPU by switching between them, but the switches occur so frequently. Thus, the user can receive an immediate response. For example, in a transaction processing, the processor executes each user program in a short burst or quantum of computation. That is, if users are present, then each user can get a time quantum. When the user submits the command, the response time is in few seconds at most.

The operating system uses CPU scheduling and multiprogramming to provide each user with a small portion of a time. Computer systems that were designed primarily as batch systems have been modified to time-sharing systems.

- Multiprogramming didn't provide the user interaction with the computer system.
- Time sharing or Multitasking is a logical extension of Multiprogramming that provides user interaction.
- There are more than one user interacting the system at the same time
- The switching of CPU between two users is so fast that it gives the impression to user that he is only working on the system but actually it is shared among different users.
- CPU bound is divided into different time slots depending upon the number of users using the system.
- Just as multiprogramming allows the processor to handle multiple batch jobs at a time, multiprogramming can also be used to handle multiple interactive jobs. In this latter case, the technique is referred to as time sharing, because processor time is shared among multiple users
- A multitasking system uses CPU scheduling and multiprogramming to provide each user with a small portion of a time shared computer. Each user has at least one separate program in memory.
- Multitasking are more complex than multiprogramming and must provide a mechanism for jobs synchronization and

communication and it may ensure that system does not go in deadlock.

Although batch processing is still in use but most of the system today available uses the concept of multitasking and Multiprogramming.

Advantages

- Provides the advantage of quick response.
- Avoids duplication of software.
- Reduces CPU idle time.

Disadvantages

- Problem of reliability.
- Question of security and integrity of user programs and data.
- Problem of data communication.

Parallel Systems

Parallel Processing Systems are designed to speed up the execution of programs by dividing the program into multiple fragments and processing these fragments simultaneously. Such systems are multiprocessor systems also known as tightly coupled systems. Parallel systems deal with the simultaneous use of multiple computer resources that can include a single computer with multiple processors, a number of computers connected by a network to form a parallel processing cluster or a combination of both.

In computers, parallel processing is the processing of program instructions by dividing them among multiple processors with the objective of running a program in less time. In the earliest computers, only one program ran at a time. A computation-intensive program that took one hour to run and a tape copying program that took one hour to run would take a total of two hours to run. An early form of parallel processing allowed the interleaved execution of both programs together. The computer would start an I/O operation, and while it was waiting for the operation to complete, it would execute the processor-intensive program. The total execution time for the two jobs would be a little over one hour.

Parallel computing is an evolution of serial computing where the jobs are broken into discrete parts that can be executed concurrently. Each part is further broken down to a series of instructions. Instructions from each part execute simultaneously on different CPUs.

Parallel systems are more difficult to program than computers with a single processor because the architecture of parallel computers varies accordingly and the processes of multiple CPUs must be coordinated and synchronized. Several models for connecting processors and memory modules exist, and each topology requires a different programming model. The three models that are most commonly used in building parallel computers include synchronous processors each with its own memory, asynchronous processors each with its own memory and asynchronous processors with a common, shared memory. Flynn has classified the computer systems based on parallelism in the instructions and in the data streams. These are:

1. Single instruction stream, single data stream (SISD).
2. Single instruction stream, multiple data stream (SIMD).
3. Multiple instruction streams, single data stream (MISD).
4. Multiple instruction stream, multiple data stream (MIMD).

Distributed Systems

Distributed systems use multiple central processors to serve multiple real-time applications and multiple users. Data processing jobs are distributed among the processors accordingly.

The processors communicate with one another through various communication lines (such as high-speed buses or telephone lines). These are referred as loosely coupled systems or distributed systems. Processors in a distributed system may vary in size and function. These processors are referred as sites, nodes, computers, and so on.

Advantages

- With resource sharing facility, a user at one site may be able to use the resources available at another.
- Speedup the exchange of data with one another via electronic mail.

- If one site fails in a distributed system, the remaining sites can potentially continue operating.
- Better service to the customers.
- Reduction of the load on the host computer.
- Reduction of delays in data processing.

Real Time Systems.

A real-time system is defined as a data processing system in which the time interval required to process and respond to inputs is so small that it controls the environment. The time taken by the system to respond to an input and display of required updated information is termed as the response time. So in this method, the response time is very less as compared to online processing.

Real-time systems are used when there are rigid time requirements on the operation of a processor or the flow of data and real-time systems can be used as a control device in a dedicated application. A real-time operating system must have well-defined, fixed time constraints, otherwise the system will fail. For example, scientific experiments, medical imaging systems, industrial control systems, weapon systems, robots, air traffic control systems, etc.

There are two types of real-time operating systems.

1. Hard real-time systems

Hard real-time systems guarantee that critical tasks complete on time. In hard real-time systems, secondary storage is limited or missing and the data is stored in ROM. In these systems, virtual memory is almost never found.

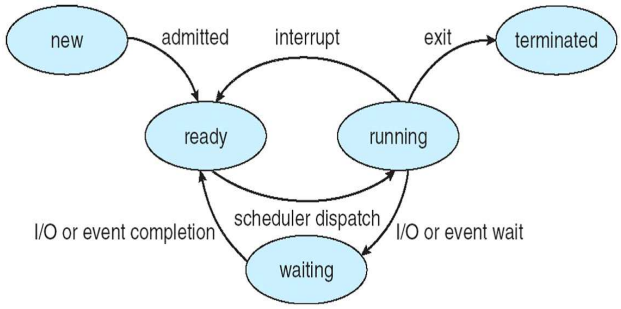
2. Soft real-time systems

Soft real-time systems are less restrictive. A critical real-time task gets priority over other tasks and retains the priority until it completes. Soft real-time systems have limited utility than hard real-time systems. For example, multimedia, virtual reality, Advanced Scientific Projects likes undersea exploration and planetary rovers, etc.

Module II
OPERATING SYSTEM

DEFINITION OF PROCESS

A program in Execution is called process. A process is an instance of a computer program that is being executed. It contains the program code and its current activity. Depending on the operating system (OS), a process may be made up of multiple threads of execution that execute instructions concurrently. A program is a passive entity while a process is an active entity.



PROCESS STATES

As a process executes, it changes state

- **New**: The process is being created
- **Running**: Instructions are being executed
- **Waiting**: The process is waiting for some event to occur
- **Ready**: The process is waiting to be assigned to a processor
- **Terminated**: The process has finished execution

Only one process is running and all other processes are in ready or waiting.

PROCESS CONTROL BLOCK

PCB is a data structure in the operating system kernel containing the information needed to manage a particular process. The PCB is a certain store that allows the operating systems to locate key information about a process. Thus, the PCB is the data structure that defines a process to the operating systems.

The PCB contains important information about the specific process including

- **Process State**: The current state of the process i.e., whether it is ready, running, waiting, or whatever.
- **Process Number**: Unique identification of the process in order to track "which is which" information.
- **Program Counter**: This register stores the next instruction to be executed.

- **CPU Registers**: MAR, MBR, PC, IR, GPR
- **Memory limits**: process stored location in memory
- **List of files**:
- **The priority of process**
- **A pointer to parent process.**
- **A pointer to child process** (if it exists).

OPERATIONS ON PROCESS

1. **Process creation**: A user requests and already running process can creates new processes. Parent process creates children processes using a system call, which, in turn create other processes, forming a tree of processes.
2. **Process preempting**: A process preempted if I/O event or timeout occurs. Then process moves from running state to ready state and CPU loads another process from ready state to running state, if available.
3. **Process blocking**: When a process needs I/O event during its execution, then process moves from running state to waiting state and dispatches another process to CPU.
4. **Process termination**: A process terminated if when a process completes its execution. Also, these events: OS, Hardware interrupt, and Software interrupt can cause termination of a process.

Process Creation

Process creation is a task of creating new processes. There are different situations in which a new process is created. There are different ways to create new process. A new process can be created at the time of initialization of operating system or when system calls such as fork () are initiated by other processes. The process, which creates a new process using system calls, is called parent process while the new process that is created is called child process. The child processes can create new processes using system calls. A new process can also create by an operating system based on the request received from the user.

The process creation is very common in running computer system because corresponding to every task that is performed there is a process associated with it. For instance, a new process is created every time a user logs on to a computer system, an application program such a MS Word is initiated, or when a document printed.

Process Preemption

Tasks are usually assigned with priorities. At times it is necessary to run a certain task that has a higher priority before another task although it is running. Therefore, the running task is interrupted for some time and resumed later when the priority task has finished its execution. This is called preemption.

Process Blocking

A blocking process is usually waiting for an event such as a semaphore being released or a message arriving in its message queue. In multitasking systems, such processes are expected to notify the scheduler with a system call that it is to wait, so that they can be removed from the active scheduling queue until the event occurs. A process that continues to run while waiting (i.e., continuously polling for the event in a tight loop) is said to be busy-waiting, which is undesirable as it wastes clock cycles which could be used for other processes.

Process Termination

Process termination is an operation in which a process is terminated after the execution of its last instruction. This operation is used to terminate or end any process. When a process is terminated, the resources that were being utilized by the process are released by the operating system. When a child process terminates, it sends the status information back to the parent process before terminating. The child process can also be terminated by the parent process if the task performed by the child process is no longer needed. In addition, when a parent process terminates, it has to terminate the child process as well became a child process cannot run when its parent process has been terminated.

The termination of a process when all its instruction has been executed successfully is called normal termination. However, there are instances when a process terminates due to some error. This termination is called as abnormal termination of a process.

PROCESS COMMUNICATION

Inter-process communication (IPC) is a set of programming interfaces that allows a programmer to coordinate activities among different program processes that can run concurrently in an operating system. This allows a program to handle many user requests at the same time. Since even a single user request may result in multiple processes running in the operating system on the user's

behalf, the processes need to communicate with each other. The IPC interfaces make this possible. Each IPC method has its own advantages and limitations so it is not unusual for a single program to use all of the IPC methods.

A mechanism through which data is shared among the process in the system is referred to as Inter-process communication. Multiple processes communicate with each other to share data and resources. A set of functions is required for the communication of process with each other. In multiprogramming systems, some common storage is used where process can share data. The shared storage may be the main memory or it may be a shared file. Files are the most commonly used mechanism for data sharing between processes. One process can write in the file while another process cam read the data for the same file.

Various techniques can be used to implement the Inter-Process Communication. There are two fundamental models of Inter-Process communication that are commonly used, these are:

1. **Shared Memory Model**
2. **Message Passing Model**

Shared Memory Model

In shared memory model. The co operating process shares a region of memory for sharing of information. Some operating systems use the supervisor call to create a share memory space. Similarly, Some operating system use file system to create RAM disk, which is a virtual disk created in the RAM. The shared files are stored in RAM disk to share the information between processes. The shared files in RAM disk are actually stored in the memory. The Process can share information by writing and reading data to the shared memory location or RAM disk.

Message Passing Model

In this model, data is shared between process by passing and receiving messages between co-operating process. Message passing mechanism is easier to implement than shared memory but it is useful for exchanging smaller amount of data.

In message passing mechanism data is exchange between processes through kernel of operating system using system calls. Message passing mechanism is particularly useful in a distributed

environment where the communicating processes may reside on different components connected by the network. For example, A data program used on the internet could be designed so that chat participants communicate with each other by exchanging messages. It must be noted that passing message technique is slower than shared memory technique.

A message contains the following information:

- Header of message that identifies the sending and receiving processes
- Block of data
- Pointer to block of data
- Some control information about the process

Typically Inter-Process Communication is based on the ports associated with process. A port represents a queue of processes. Ports are controlled and managed by the kernel. The processes communicate with each other through kernel.

In message passing mechanism, two operations are performed. Theses are sending message and receiving message. The function send() and receive() are used to implement these operations. Supposed P1 and P2 want to communicate with each other. A communication link must be created between them to send and receive messages. The communication link can be created using different ways. The most important methods are:

- Direct model
- Indirect model
- Buffering

COMMUNICATION IN CLIENT SERVER SYSTEM

Client-server concept underpins distributed systems over a couple of decades. There are two counterparts in the concept: a client and a server. In practice there are often multiple clients and single server. Clients start communication by sending requests to the server, the server handles them and usually returns responses back. Client processes often do not live long, while server process, which is sometimes called daemon, live till OS shutdown.

Sockets

IPC with sockets is very common in distributed systems. In nutshell, a socket is a pair of an IP address and a port number. For two processes to communicate, each of them needs a socket.

When server daemon is running on a host, it is listening to its port and handles all requests sent by clients to the port on the host (server socket). A client must know IP and port of the server (server socket) to send a request to it. Client's port is often provided by OS kernel when client starts communication with the server and is freed when communication is over.

Although communication using sockets is common and efficient, it is considered low level, because sockets only allow to transfer unstructured stream of bytes between processes. It is up to client and server applications to impose a structure on the data passed as byte stream.

Remote Procedure Calls

RPC is a higher level communication method. It was designed to mimic procedure call mechanism, but execute it over network. RPC is conceptually similar to message passing IPC and is usually built on top of socket communication. In contrast to IPC messages, RPC messages are well structured. Each message includes information about function to be executed and the parameters to be passed to that function. When the function is executed, a response with output is sent back to the requester in a separate message.

RPC hides the details of communication by providing a stub on the client side. When client needs to invoke a remote procedure, it invokes the stub and pass it the parameters. The stub marshals the parameters and sends a message to RPC daemon running on the server. RPC daemon (a similar stub on the server side) receives the message and invokes the procedure on the server. Return values are passed back to the client using the same technique.

Pipes

Pipe is one of the oldest and simplest IPC methods, that appeared in early UNIX systems. A pipe is an IPC abstraction with two endpoints, similar to a physical pipe. Usually one process puts data to one end of the pipe and another process consumes them from the other one.

Ordinary pipes

Ordinary pipes are unidirectional, i.e. allow only one-way communication. They implement standard producer-consumer mechanism, where one process writes to the pipe and another one reads from it. For two-way communication two pipes are needed. Ordinary pipes require a parent–child relationship between communicating processes, because a pipe can only be accessed from process that created or inherited it. Parent process creates a pipe and uses it to communicate with a child created via fork(). Once communication is over and processes terminated, the ordinary pipe ceases to exist.

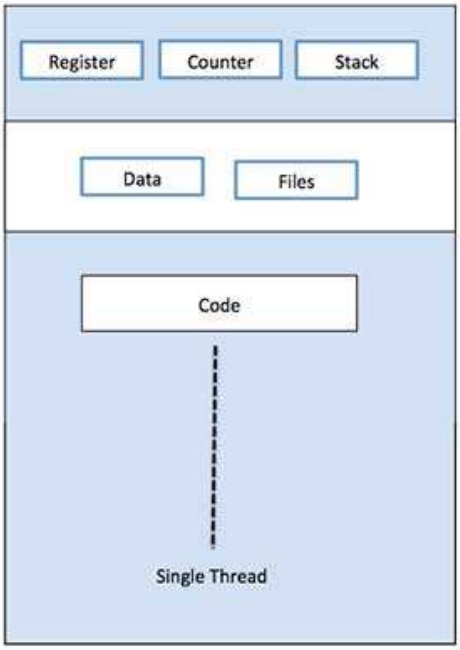
Named pipes

Named pipes are more powerful. They do not require parent–child relationship and can be bidirectional. Once a named pipe is created, multiple non related processes can communicate over it. Named pipe continues to exist after communicating processes have terminated. It must be explicitly deleted when not required anymore.

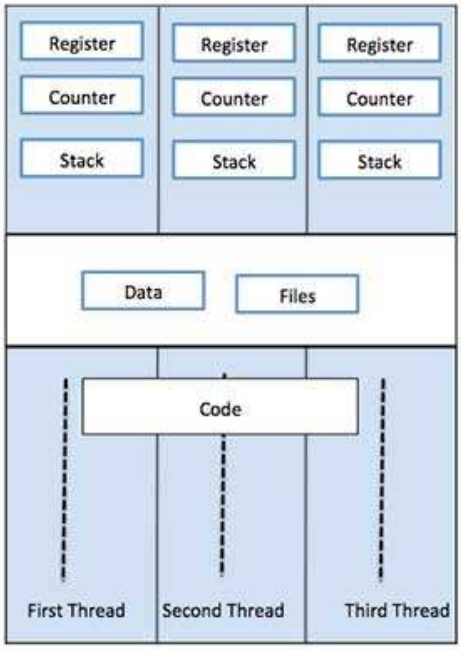
BASIC CONCEPTS OF THREADS

A thread is a flow of execution through the process code, with its own program counter that keeps track of which instruction to execute next, system registers which hold its current working variables, and a stack which contains the execution history. A thread shares with its peer threads few information like code segment, data segment and open files. When one thread alters a code segment memory item, all other threads see that.

A thread is also called a lightweight process. Threads provide a way to improve application performance through parallelism. Threads represent a software approach to improving performance of operating system by reducing the overhead thread is equivalent to a classical process. Each thread belongs to exactly one process and no thread can exist outside a process. Each thread represents a separate flow of control. Threads have been successfully used in implementing network servers and web server. They also provide a suitable foundation for parallel execution of applications on shared memory multiprocessors. The following figure shows the working of a single-threaded and a multithreaded process.



Single Process P with single thread



Single Process P with three threads

Difference between Process and Thread

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

Advantages of Thread

- Threads minimize the context switching time.
- Use of threads provides concurrency within a process.
- Efficient communication.
- It is more economical to create and context switch threads.
- Threads allow utilization of multiprocessor architectures to a greater scale and efficiency.

Types of Thread

Threads are implemented in following two ways −

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

User Level Threads

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

Advantages

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

Disadvantages

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

Kernel Level Threads

In this case, thread management is done by the Kernel. There is no thread management code in the application area. Kernel threads are supported directly by the operating system. Any application can be programmed to be multithreaded. All of the threads within an application are supported within a single process. The Kernel maintains context information for the process as a whole and for individuals threads within the process. Scheduling by the Kernel is done on a thread basis. The Kernel performs thread creation, scheduling and management in Kernel space. Kernel threads are generally slower to create and manage than the user threads.

Advantages

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

Disadvantages

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

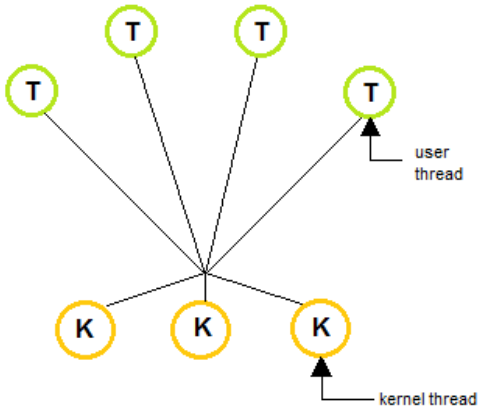
Multithreading Models

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

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

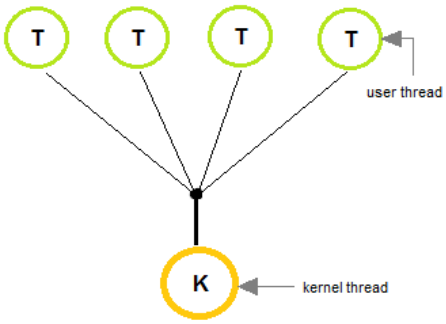
Many to Many Model

The many-to-many model multiplexes any number of user threads onto an equal or smaller number of kernel threads.



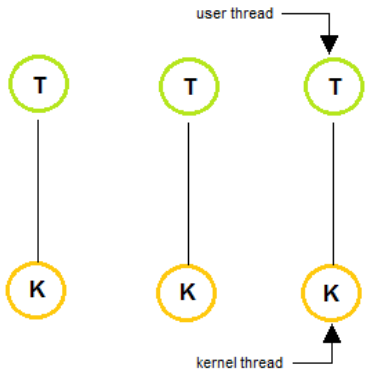
The following diagram shows the many-to-many threading model where 6 user level threads are multiplexing with 6 kernel level threads. In this model, developers can create as many user threads as necessary and the corresponding Kernel threads can run in parallel on a multiprocessor machine. This model provides the best accuracy on concurrency and when a thread performs a blocking system call, the kernel can schedule another thread for execution.

Many to One Model



Many-to-one model maps many user level threads to one Kernel-level thread. Thread management is done in user space by the thread library. When thread makes a blocking system call, the entire process will be blocked. Only one thread can access the Kernel at a time, so multiple threads are unable to run in parallel on multiprocessors. If the user-level thread libraries are implemented in the operating system in such a way that the system does not support them, then the Kernel threads use the many-to-one relationship modes.

One to One Model



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

Difference between User-Level & Kernel-Level Thread

Sl.No.	User-Level Threads	Kernel-Level Thread
1	User-level threads are faster to create and manage.	Kernel-level threads are slower to create and manage.
2	Implementation is by a thread library at the user level.	Operating system supports creation of Kernel threads.
3	User-level thread is generic and can run on any operating system.	Kernel-level thread is specific to the operating system.
4	Multi-threaded applications cannot take advantage of multiprocessing.	Kernel routines themselves can be multithreaded.

CONCURRENCY

Concurrent processing is a computing model in which multiple processors execute instructions simultaneously for better performance. Concurrent means something that happens at the same time as something else. Tasks are broken down into subtasks that are then assigned to separate processors to perform simultaneously. Concurrent processing is sometimes said to be synonymous with parallel processing.

Concurrency is the tendency for things to happen at the same time in a system. Concurrency is a natural phenomenon, of course. In the real world, at any given time, many things are happening simultaneously. When we design software to monitor and control real-world systems, we must deal with this natural concurrency. When dealing with concurrency issues in software systems, there are generally two aspects that are important: being able to detect and respond to external events occurring in a random order, and ensuring that these events are responded to in some minimum required interval. Concurrency cannot be avoided because:

- Users are concurrent - a person can handle several tasks at once and expects the same from a computer.
- Multiprocessors are becoming more prevalent.
- The Internet is perhaps a huge multiprocessor.
- A distributed system (client/server system) is naturally concurrent.
- A windowing system is naturally concurrent.

I/O is often slow because it involves slow devices such as disks, printers; many network operations are essentially (slow) I/O operations. When doing I/O it is helpful to handle the I/O concurrently with other work. Whenever concurrency is involved certain issues, discussed below, arise. An understanding of these issues is important when:

- writing an operating system.
- when interacting with the kernel, for example, when performing I/O.
- when generating multiple processes, for example, with forks and pipelines.
- when using multiple threads.

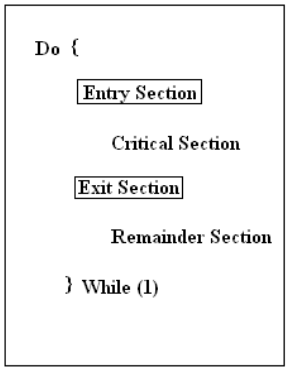
Concurrency issues:

- **Atomic**. An operation is atomic if the steps are done as a unit. Operations that are not atomic, but interruptible and done by multiple processes can cause problems. For example, an lseek followed by a write is not atomic. A process is likely to lose its time quantum between the lseek (a slow operation if the distance seeked is large!) and the write. If another process has the file open and does a write then the result is not what is intended.
- **Race conditions**. A race condition occurs if the outcome depends on which of several processes gets to a point first. For example, fork( ) can generate a race condition if the result depends on whether the parent or the child process runs first. Other race conditions can occur if two processes are updating a global variable.
- **Blocking and starvation**. While neither of these problems is unique to concurrent processes, their effects must be carefully considered. Processes can block waiting for resources. A process could be blocked for a long period of time waiting for input from a terminal. If the process is required to periodically update some data, this would be very undesirable. Starvation occurs when a process does not obtain sufficient CPU time to make meaniful progress.
- **Deadlock**. Deadlock occurs when two processes are blocked in such a way that neither can proceed. The typical occurrence is where two processes need two non-shareable resources to proceed but one process has acquired one resource and the other has acquired the other resource. Acquiring resources in a specific order can resolve some deadlocks.

PRINCIPLES OF CONCURRENCY

Concurrency is the tendency for things to happen at the same time in a system. It also refers to techniques that make program more usable. Concurrency can be implemented and is used a lot on single processing units, nonetheless it may benefit from multiple processing units with respect to speed. If an operating system is called a multi-tasking operating system, this is a synonym for supporting concurrency. If we can load multiple documents simultaneously in the tabs of our browser and we can still open menus and perform more actions, this is concurrency. If we run distributed-net computations in the background, that is concurrency.

MUTUAL EXCLUSION AND CRITICAL SECTION



Consider a system consisting of n processes {P0, P1, …, Pn-1}. Each process has a segment of code, called a critical section, in which the process may be changing common variables, updating a table, writing a file, and so on. When one process is executing in its critical section, no other process is to be allowed to execute in its critical section. Thus, the execution of critical sections by the processes is mutually exclusive in time. Each process must request permission to enter its critical section. The section of code implementing this request is the entry section. The critical section may be followed by an exit section. The remaining code is the remainder section. A picture (right side) showing general structure of a process Pi.

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

- **Mutual Exclusion**: If process Pi is executing in its critical section, then no other processes can be executing in their critical sections.

- **Progress**: If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder section can participate in the decision on which will enter its critical section next, and this selection cannot be postponed indefinitely.

- **Bounded Waiting**: There exists a bound on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

SEMAPHORE

To generalize to more complex problems the solutions to the critical-section problem are not easy. To overcome this difficulty, we can use a synchronization tool called a semaphore. A semaphore S is an integer variable that, apart from initialization, is accessed only through two standard atomic operations: wait and signal. These operations were originally termed P (for wait, to test) and V (for signal, to increment). The classical definition of wait in pseudo code is:

```
wait(S) {
while (S 5 0)
; // no-op
s--;
}
```

The classical definitions of signal in pseudo code are:

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

Modifications to the integer value of the semaphore in the wait and signal operations must be executed indivisibly. That is, when one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value. In addition, in the case of the wait (S), the testing of the integer value of S (S 5 O), and its possible modification (S--), must also be executed without interruption.

DEAD LOCK

In a multiprogramming environment, several processes may compete for a finite number of resources. A process requests resources; if the resources are not available at that time, the process enters a wait state. Waiting processes may never again change state, because the resources they have requested are held by other waiting processes. This situation is called a deadlock.

Necessary Conditions

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

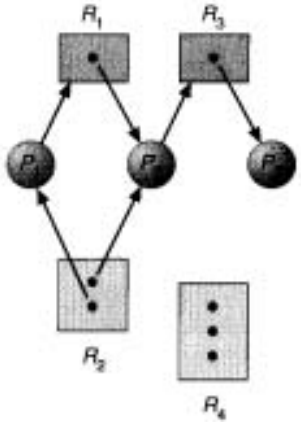
- **Mutual exclusion**: At least one resource must be held in a non-sharable mode; that is, only one process at a time can use the resource. If another process requests that resource, the requesting process must be delayed until the resource has been released.

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

- **No preemption**: Resources cannot be preempted; that is, resources can be released only voluntarily by the process holding it, after that process has completed its task.

- **Circular wait**: A set {P0, P1, ..., Pn) of waiting processes must exist such that P0 is waiting for a resource that is held by PI, PI is waiting for a resource that is held by P2, ..., Pn-1 is waiting for a resource that is held by P,, and P, is waiting for a resource that is held by P0.

Resource-Allocation Graph (RAG)



Deadlocks can be described more precisely in terms of a directed graph called a system resource-allocation graph. This graph consists of a set of vertices V and a set of edges E. The set of vertices V is partitioned into two different types of nodes P = {P1, P2, ..., Pn}, the set consisting of all the active processes in the system, and R = {R1, R2, ..., Rm), the set consisting of all resource types in the system.

The resource-allocation graph show depicts the following situation.

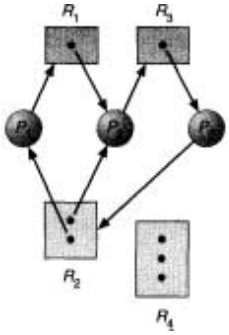
The sets P, R, and E:

P = {P1, P2, P3}
R = {R1, R2, R3, R4}
E = {P1 R1, P2 R3, R1 P2, R2 P2, R2 P1, R3 P3}

Resource instances:

One instance of resource type R1
Two instances of resource type R2
One instance of resource type R3
Three instances of resource type R4

Process states:



Process P1 is holding an instance of resource type R2, and is waiting for an instance of resource type R1
Process P2 is holding an instance of R1 and R2, and is waiting for an instance of resource type R3.
Process P3 is holding an instance of R3.

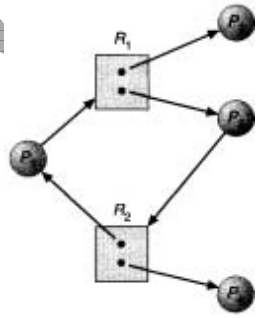
A request edge P3 R2 is added to the graph. At this

point, two minimal cycles exist in the system. They are:

$$P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$$
$$P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_2$$

The processes P1, P2, and P3 are deadlocked. Process P2 is waiting for the resource R3, which is held by process P3. Process P3, on the other hand, is waiting for either process P1 or process P2 to release resource R2. In addition, process P1 is waiting for process P2 to release resource R1. The Resource allocation graph with a deadlock in the figure.

Now consider the resource-allocation graph in following Figure. In this example, we also have a cycle. However, there is no deadlock. Observe that process P4 may release its instance of resource type R2. That resource can then be allocated to P3, breaking the cycle. In summary, if a resource-allocation graph does not have a cycle, then the system is not in a deadlock state. On the other hand, if there is a cycle, then the system may or may not be in a deadlock state.



HANDLING DEADLOCK

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

- We can use a protocol to prevent or avoid deadlocks, ensuring that the system will never enter a deadlock state.
- We can allow the system to enter a deadlock state, detect it, and recover.
- We can ignore the problem altogether, and pretend that deadlocks never occur in the system. This solution is used by most operating systems, including UNIX.

DEAD LOCK PREVENTION

Prevent deadlocks by restraining how requests can be made. For a deadlock to occur, each of the four necessary conditions must hold. By ensuring that at least one of these conditions cannot hold, we can prevent the occurrence of a deadlock.

Mutual Exclusion

The mutual-exclusion condition must hold for non-sharable resources. For example, a printer cannot be simultaneously shared by several processes. Sharable resources, do not require mutually exclusive access, and thus cannot be involved in a deadlock. Read-only

files are a good example of a sharable resource. If several processes attempt to open a read-only file at the same time, they can be granted simultaneous access to the file. A process never needs to wait for a sharable resource. In general, however, we cannot prevent deadlocks by denying the mutual-exclusion condition: Some resources are intrinsically non-sharable.

Hold and Wait

To ensure that the hold-and-wait condition never occurs in the system, we must guarantee that, whenever a process requests a resource, it does not hold any other resources. One protocol that can be used requires each process to request and be allocated all its resources before it begins execution.

An alternative protocol allows a process to request resources only when the process has none. A process may request some resources and use them. Before it can request any additional resources, however, it must release all the resources that it is currently allocated.

No Preemption

The third necessary condition is that there is no preemption of resources that have already been allocated. If a process is holding some resources and requests another resource that cannot be immediately allocated to it (that is, the process must wait), then all resources currently being held are preempted. If a process requests some resources, we first check whether they are available. If they are, we allocate them. If they are not available, we check whether they are allocated to some other process that is waiting for additional resources. If so, we preempt the desired resources from the waiting process and allocate them to the requesting process.

Circular Wait

The fourth for deadlocks is the circular-wait condition. One way to ensure that this condition never holds is to impose a total ordering of all resource types, and to require that each process requests resources in an increasing order of enumeration. We can require that, whenever a process requests an instance of resource type Rj, it has released any resources Ri such that F(Ri) → F(Rj). Possible side effects of preventing deadlocks by this method, however, are low device utilization and reduced system throughput.

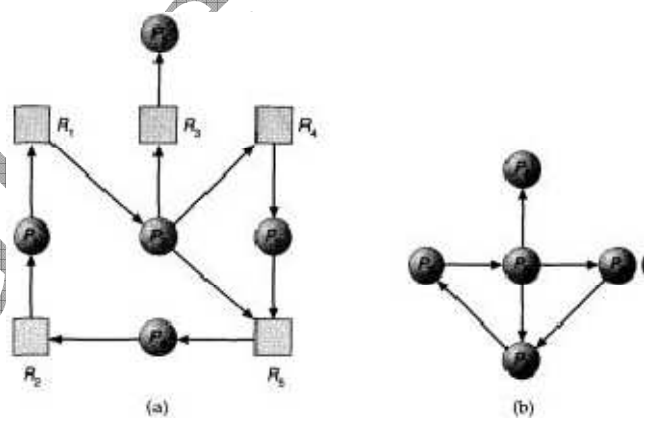
DEAD LOCK DETECTION

If a system does not employ either a deadlock-prevention or a deadlock avoidance algorithm, then a deadlock situation may occur. In this environment, the system must provide:

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

Single Instance of Each Resource Type

If all resources have only a single instance, then we can define a deadlock detection algorithm that uses a variant of the resource-allocation graph, called a wait-for graph. We obtain this graph from the resource-allocation graph by removing the nodes of type resource and collapsing the appropriate edges.



More clearly, an edge from Pi to Pi in a wait-for graph implies that process Pi is waiting for process Pi to release a resource that Pi needs. An edge Pi → Pi exists in a wait-for graph if and only if the corresponding resource allocation graph contains two edges Pi → Rq and Rq → Pj for some resource Rq. For example, in Figure (a) Resource-allocation graph. (b) Corresponding wait-for graph, we present a resource-allocation graph and the corresponding wait-for graph. As before, a deadlock exists in the system if and only if the wait-for graph contains a cycle. To detect deadlocks, the system needs to maintain the wait-for graph and periodically to invoke an algorithm that searches for a cycle in the graph.

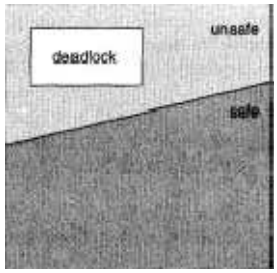
If deadlocks occur frequently, then the detection algorithm should be invoked frequently. Resources allocated to deadlocked processes will be idle until the deadlock can be broken. In addition, the number of processes involved in the deadlock cycle may grow. Deadlocks occur only when some process makes a request that cannot be granted immediately. This request may be the final request that completes a chain of waiting processes.

DEAD LOCK AVOIDANCE

The method for avoiding deadlocks is to require additional information about how resources are to be requested. Each process declares the maximum number of resources of each type that it may need. Given a priori information about the maximum number of resources of each type that may be requested for each process, it is possible to construct an algorithm that ensures that the system will never enter a deadlock state. This algorithm defines the deadlock-avoidance approach.

A deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that a circular wait condition can never exist. The resource-allocation state is defined by the number of available and allocated resources, and the maximum demands of the processes.

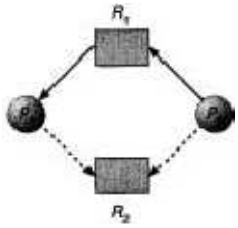
Safe State



A state is safe if the system can allocate resources to each process (up to its maximum) in some order and still avoid a deadlock. More formally, a system is in a safe state only if there exists a safe sequence. The Safe, unsafe, and deadlock state spaces are shows in the figure. A safe state is not a deadlock state. Conversely, a deadlock state is an unsafe state. Not all unsafe states are deadlocks. An unsafe state may lead to a deadlock.

Resource-Allocation Graph Algorithm

If we have a resource-allocation system with only one instance of each resource type, a variant of the resource-allocation graph defined in previous section can be used for deadlock avoidance. In addition to the request and assignment edges, we introduce a new type of edge, called a claim edge. A claim edge Pi→Rj indicates that process Pi may request resource Rj at some time in the future. This edge resembles a request edge in direction, but is represented by a dashed line. When process Pi requests resource Rj, the claim edge Pi→Rj is converted to a request edge. Similarly, when a resource Rj is released by Pi the assignment edge Rj →Pi is reconverted to a claim edge Pi → Rj. We note that the resources must be claimed a priori in the system. That is, before process Pi starts executing, all its claim edges must already appear in the resource-



allocation graph. We can relax this condition by allowing a claim edge Pi→Rj to be added to the graph only if all the edges associated with process Pi are claim edges.

Suppose that process Pi requests resource Rj. The request can be granted only if converting the request edge Pi → Rj to an assignment edge Rj → Pi does not result in the formation of a cycle in the resource-allocation graph. Note that we check for safety by using a cycle-detection algorithm. An algorithm for detecting a cycle in this graph requires an order of n2 operations, where n is the number of processes in the system.

If no cycle exists, then the allocation of the resource will leave the system in a safe state. If a cycle is found, then the allocation will put the system in an unsafe state. Therefore, process Pi will have to wait for its requests to be satisfied.

Banker's Algorithm

The resource-allocation graph algorithm is not applicable to a resource allocation system with multiple instances of each resource type. The next is applicable to such a system, but is less efficient than the resource-allocation graph scheme. This algorithm is commonly known as the banker's algorithm. The name was chosen because this algorithm could be used in a banking system to ensure that the bank never allocates its available cash such that it can no longer satisfy the needs of all its customers.

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

We need the following data structures:

- **Available**: A vector of length m indicates the number of available resources of each type. If Available[j] = k, there are k instances of resource type Rj available.

- **Max**: An n, x m matrix defines the maximum demand of each process. If Max[i,j] = k, then process Pi may request at most k instances of resource type Ri.

- **Allocation**: An n x m matrix defines the number of resources of each type currently allocated to each process. If Allocation[i,j] = k, then process Pi is currently allocated k instances of resource type Rj.

- **Need**: An n x m matrix indicates the remaining resource need of each process. If Need[i,j] = k, then process Pi may need k more instances of resource type Ri to complete its task. Note that Need[i,j] = Max[i,j] - Allocafion[i,j].

Safety Algorithm

1. Let Work and Finish be vectors of length m and n, respectively.

```
Initialize Work := Available   and
Finish[i] :=false for i = 1,2, ..., n.
```

2. Find an i such that both

```
a.  Finish[i] =false
b.  Needi 5 Work.
```

If no such i exists, go to step 4.

3. Work := Work + Allocation

```
Finish[i] := true
go to step 2.
```

4. If Finish[i] = true for all i, then the system is in a safe state. This algorithm may require an order of m x n2 operations to decide whether a state is safe.

Resource-Request Algorithm

Let Requesti be the request vector for process Pi. If Request;[j] = k, then process Pi wants k instances of resource type Rj. When a request for resources is made by process Pi, the following actions are taken:

1. If Requesti 5 Needi, go to step 2. Otherwise, raise an error condition, since the process has exceeded its maximum claim.
2. If Requesti 5 Available, go to step 3. Otherwise, Pi must wait, since the resources are not available.
3. Have the system pretend to have allocated the requested resources to process Pi by modifying the state as follows:

```
Available := Available - Request,;
Allocationi := Allocation; + Request;;
Needi := Needi - Requesti;
```

If the resulting resource-allocation state is safe, the transaction is completed and process Pi is allocated its resources. However, if the new state is unsafe, then Pi must wait for Requesti and the old resource-allocation state is restored.

RECOVERY FROM DEADLOCK

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

Process Termination

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

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

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

Resource Preemption

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

- **Selecting a victim**: Which resources and which processes are to be preempted? As in process termination, we must determine the order of pre-emption to minimize cost. Cost factors may include such parameters as the number of resources a deadlock process is holding, and the amount of time a deadlocked process has thus far consumed during its execution.

- **Rollback**: If we preempt a resource from a process, what should be done with that process? Clearly, it cannot continue with its normal execution; it is missing some needed resource. We must roll back the process to some safe state, and restart it from that state.

Since, in general, it is difficult to determine what a safe state is, the simplest solution is a total rollback: Abort the process and then restart it. However, it is more effective to roll back the process only as far as necessary to break the deadlock. On the other hand, this method requires the system to keep more information about the state of all the running processes.

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

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

MODULE II
LINUX SHELL PROGRAMMING

Introduction

Shell programming is a basic skill that every UNIX System administrator should have. The Systems Administrator must be able to read and write shell programs because

- There are many tasks that can and should be quickly automated by using shell programs
- Many software products come with install scripts that have to be modified for our system before they will work.

A Kernel is at the nucleus of a computer. It makes the communication between the hardware and software possible. While the Kernel is the innermost part of an operating system, a shell is the outermost one. A shell in a Linux operating system takes input from we in the form of commands, processes it, and then gives an output. It is the interface through which a user works on the programs, commands, and scripts. A shell is accessed by a terminal which runs it.

When we run the terminal, the Shell issues a command prompt (usually $), where we can type our input, which is then executed when we hit the Enter key. The output or the result is thereafter displayed on the terminal. The Shell wraps around the delicate interior of an Operating system protecting it from accidental damage. Hence the name Shell.

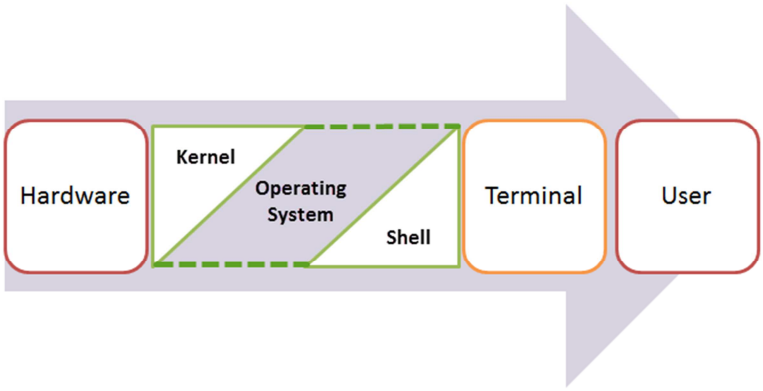
If we are using any major operating system we are indirectly interacting to shell. If we are running Ubuntu, Linux Mint or any other Linux distribution, we are interacting to shell every time we use terminal. In this article I will discuss about linux shells and shell scripting so before understanding shell scripting we have to get familiar with following terminologies –

- Kernel
- Shell
- Terminal

What is Kernel

The kernel is a computer program that is the core of a computer's operating system, with complete control over everything in the system. It manages following resources of the Linux system –

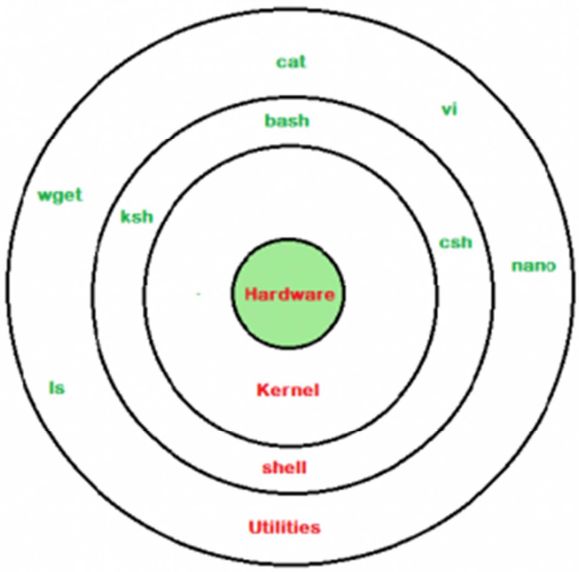
- File management
- Process management
- I/O management
- Memory management
- Device management etc.



It is often mistaken that Linus Torvalds has developed Linux OS, but actually he is only responsible for development of Linux kernel. Complete Linux system = Kernel + GNU system utilities and libraries + other management scripts + installation scripts.

What is Shell

A shell is special user program which provide an interface to user to use operating system services. Shell accept human readable commands from user and convert them into something which kernel can understand. It is a command language interpreter that execute commands read from input devices such as keyboards or from files. The shell gets started when the user logs in or start the terminal.



Linux Shell

Shell is broadly classified into two categories –

1. Command Line Shell
2. Graphical shell

Command Line Shell

Shell can be accessed by user using a command line interface. A special program called Terminal in linux/macOS or Command Prompt in Windows OS is provided to type in the human readable commands such as "cat", "ls" etc. and then it is being execute.

Graphical Shells

Graphical shells provide means for manipulating programs based on graphical user interface (GUI), by allowing for operations such as opening, closing, moving and resizing windows, as well as switching focus between windows. Window OS or Ubuntu OS can be considered as good example which provide GUI to user for interacting with program. User do not need to type in command for every actions.

There are several shells are available for Linux systems like –

- **BASH (Bourne Again SHell)** – It is most widely used shell in Linux systems. It is used as default login shell in Linux systems and in macOS. It can also be installed on Windows OS.
- **CSH (C SHell)** – The C shell's syntax and usage are very similar to the C programming language.
- **KSH (Korn SHell)** – The Korn Shell also was the base for the POSIX Shell standard specifications etc.

Each shell does the same job but understands different commands and provide different built in functions.

SHELLS AVAILABLE IN UNIX

A shell is a user program that allows the user to specify operations in a certain sequence. As we discussed in last post about what is shell in UNIX. Now we will discuss about all different type of shell available in UNIX. If all shell available on our system then we can switch between different shells. Following are different types of shells –

1. **Bourne Shell (sh)** – Bourne Shell is the original unix shell developed at AT&T by Stephen Bourne. Bourne shell also named as (sh) programming name. It used the symbol $. Bourne shell's family is bourne, korn shells, bash and zsh .
2. **Korn shell (ksh)** – korn shell is the unix shell developed by David korn of Bell labs. Is is considered as the family member of Bourne shell as it uses the $ symbol of Bourne shell. It is also names as ksh programmatic ally and it most widely used shell.
3. **Bourne Again Shell (bash)** – It is the free version of Bourne shell and comes with all UNIX/Linux systems as free with some additional features like command line editing. Its program name is bash. It can read commands from file called scripts.

Like all Unix shells it supports the following:

- File name wildcarding
- Piping
- Hear documents
- Command execution
- Variables and control structures for condition testing and iteration

4. **C Shell (sh)** – C shell is the UNIX shell created by Bill joy at California university as an alternative to Bourne shell – Unix original shell. C shell

along with Bourne and Korn, are there most popular and commonly used shells. csh is the program name for C shell.

5. **Tab C Shell (tcsh)** – It is the family member of C shell with additional features like enhanced history substitution to reuse commands, spelling correction and word completion.

Hello World Example
Create a file first.sh

```
#!/bin/sh
echo Hello World
```

Now run –

```
./first.sh
```

Output will be – Hello World

Working with VI editor – VI is main editor to work on unix systems. There are 3 main command mode on vi editor.
Command Mode – Keys works as command like insert, delete, moving to new line etc. We can not edit or type in command mode.
Insert Mode – To write anything in script, press I or A for insert mode.
Execution Mode – This mode is used to execute that we have done. Like to save changes, first press escape key then type colon and wq.
Vi editor saving and quitting commands:
:w -Save the contents of the file.
:q – Quit from vi editor.
:q! -quit from vi editor by discarding any changes.
:wq -Save the file and quit from the vi editor.

BASH: SPECIAL CHARACTERS
Here are some of the more common special characters uses:

Character	Description
" "	Whitespace — this is a tab, newline, vertical tab, form feed, carriage return, or space. Bash uses whitespace to determine where words begin and end. The first word is the command name and additional words become arguments to that command.
$	Expansion — introduces various types of expansion: parameter expansion (e.g. $var or ${var}), command substitution (e.g. $(command)), or arithmetic expansion (e.g. $((expression))). More on expansions later.
''	Single quotes — protect the text inside them so that it has a literal meaning. With them, generally any kind of interpretation by Bash is ignored: special characters are passed over and multiple words are prevented from being split.
""	Double quotes — protect the text inside them from being split into multiple words or arguments, yet allow substitutions to

	occur; the meaning of most other special characters is usually prevented.
\	Escape — (backslash) prevents the next character from being interpreted as a special character. This works outside of quoting, inside double quotes, and generally ignored in single quotes.
#	Comment — the # character begins a commentary that extends to the end of the line. Comments are notes of explanation and are not processed by the shell.
=	Assignment -- assign a value to a variable (e.g. logdir=/var/log/myprog). Whitespace is not allowed on either side of the = character.
[[ ]]	Test — an evaluation of a conditional expression to determine whether it is "true" or "false". Tests are used in Bash to compare strings, check the existence of a file, etc. More of this will be covered later.
!	Negate — used to negate or reverse a test or exit status. For example: ! grep text file; exit $?.
>, >>, <	Redirection — redirect a command's output or input to a file. Redirections will be covered later.
|	Pipe — send the output from one command to the input of another command. This is a method of chaining commands together. Example: echo "Hello beautiful." | grep -o beautiful.
;	Command separator — used to separate multiple commands that are on the same line.
{ }	Inline group — commands inside the curly braces are treated as if they were one command. It is convenient to use these when Bash syntax requires only one command and a function doesn't feel warranted.
( )	Subshell group — similar to the above but where commands within are executed in a subshell (a new process). Used much like a sandbox, if a command causes side effects (like changing variables), it will have no effect on the current shell.
(( ))	Arithmetic expression — with an arithmetic expression, characters such as +, -, *, and / are mathematical operators used for calculations. They can be used for variable assignments like ((a = 1 + 4)) as well as tests like if ((a < b)). More on this later.
$(( ))	Arithmetic expansion — Comparable to the above, but the expression is replaced with the result of its arithmetic evaluation. Example: echo "The average is $(((a+b)/2))".
*, ?	Globs -- "wildcard" characters which match parts of filenames

	(e.g. ls *.txt).
~	Home directory — the tilde is a representation of a home directory. When alone or followed by a /, it means the current user's home directory; otherwise, a username must be specified (e.g. ls ~/Documents; cp ~john/.bashrc .).
&	Background -- when used at the end of a command, run the command in the background (do not wait for it to complete).

Examples:

$ echo "I am $LOGNAME"
I am lhunath
$ echo 'I am $LOGNAME'
I am $LOGNAME
$ # boo
$ echo An open\ \ \ space
An open space
$ echo "My computer is $(hostname)"
My computer is Lyndir
$ echo boo > file
$ echo $((5 + 5))
10
$ ((5 > 0)) && echo "Five is greater than zero."
Five is greater than zero.

GETTING HELP

Type help after almost any command to bring up a help menu for that command:

help:- The help command provides information on built-in commands.

help pwd:- At the prompt, enter 'help' followed by the command we wish to learn more about.

help –d pwd:- With the '-d' option, the help command will only return a short description of the specified command.

help –s pwd:- With the -s option, help will return a short usage synopsis for the specified command.

help –m pwd:- If we prefer the look of manpages, then we'll be happy to know that the '-m' option formats the help command output as a pseudo-manpage.

MAN PAGES

Type man before almost any command to bring up a manual for that command. The man command in Linux is used to display the user manual of any command that we can run on the terminal. It provides a detailed view of the command which includes NAME, SYNOPSIS, DESCRIPTION, OPTIONS, EXIT STATUS, RETURN VALUES, ERRORS, FILES, VERSIONS, EXAMPLES, and AUTHORS.

Syntax :

```
$man [OPTION]... [COMMAND NAME]...
```

Example

```
$ man printf
$ man 2 intro
$ man -a intro
$ man -k cd
$ man -f ls
```

Advantages of shell scripts

- The command and syntax are exactly the same as those directly entered in command line, so programmer do not need to switch to entirely different syntax
- Writing shell scripts are much quicker
- Quick start
- Interactive debugging etc.

Disadvantages of shell scripts

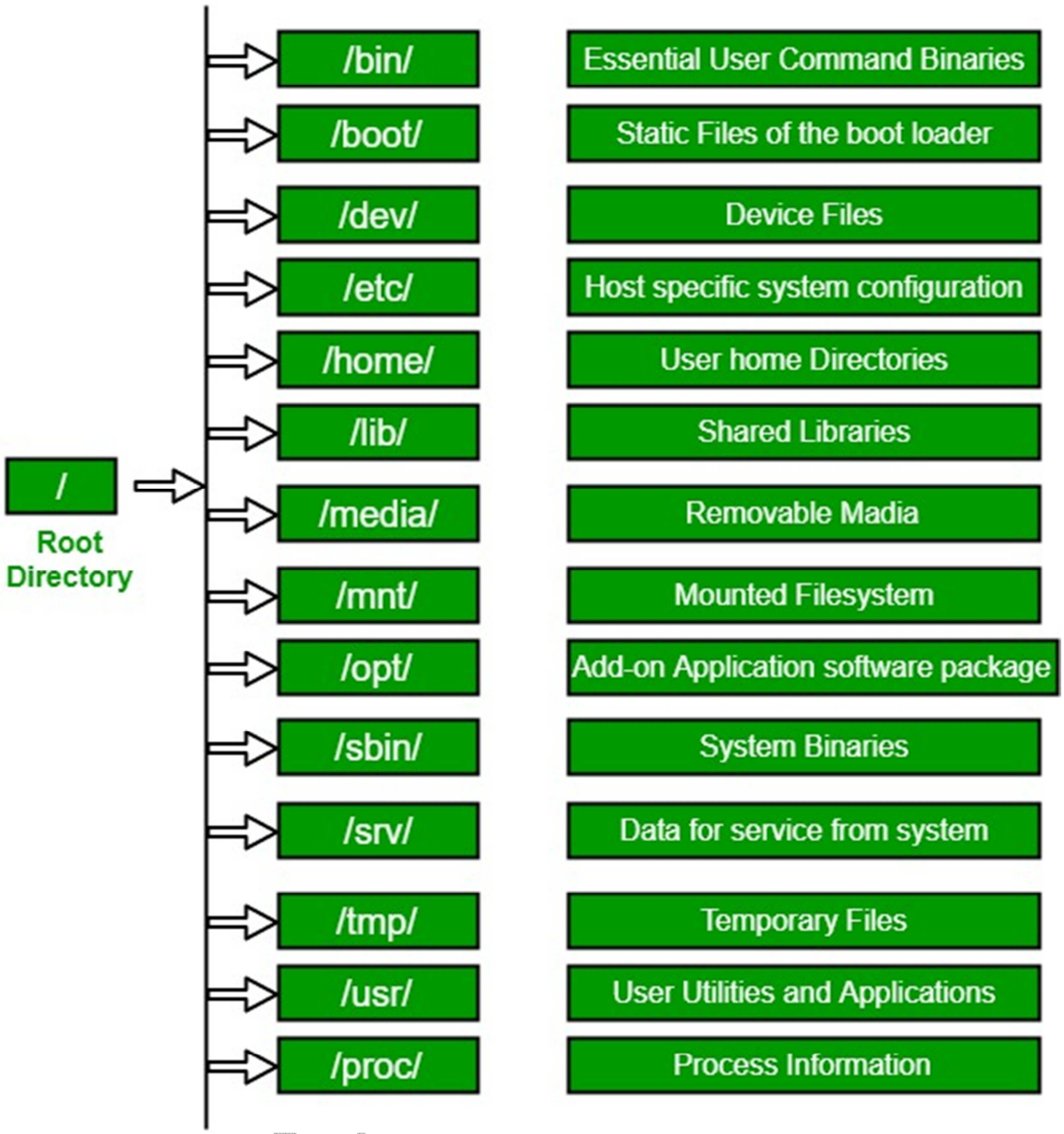
- Prone to costly errors, a single mistake can change the command which might be harmful
- Slow execution speed
- Design flaws within the language syntax or implementation
- Not well suited for large and complex task
- Provide minimal data structure unlike other scripting languages. etc

LINUX DIRECTORY LAYOUT

The Linux File Hierarchy Structure or the File system Hierarchy Standard (FHS) defines the directory structure and directory contents in Unix-like operating systems. It is maintained by the Linux Foundation.

- In the FHS, all files and directories appear under the root directory /, even if they are stored on different physical or virtual devices.
- Some of these directories only exist on a particular system if certain subsystems, such as the X Window System, are installed.
- Most of these directories exist in all UNIX operating systems and are generally used in much the same way; however, the descriptions here are those used specifically for the FHS, and are not considered authoritative for platforms other than Linux.

Each of the above directories (which is a file, at the first place) contains important information, required for booting to device drivers, configuration files, etc. Describing briefly the purpose of each directory, we are starting hierarchically.



- /**bin** : All the executable binary programs (file) required during booting, repairing, files required to run into single-user-mode, and other important, basic commands viz., cat, du, df, tar, rpm, wc, history, etc.
- /**boot** : Holds important files during boot-up process, including Linux Kernel.
- /**dev** : Contains device files for all the hardware devices on the machine e.g., cdrom, cpu, etc
- /**etc** : Contains Application's configuration files, startup, shutdown, start, stop script for every individual program.
- /**home** : Home directory of the users. Every time a new user is created, a directory in the name of user is created within home directory which contains other directories like Desktop, Downloads, Documents, etc.
- /**lib** : The Lib directory contains kernel modules and shared library images required to boot the system and run commands in root file system.

- /**lost+found** : This Directory is installed during installation of Linux, useful for recovering files which may be broken due to unexpected shut-down.
- /**media** : Temporary mount directory is created for removable devices viz., media/cdrom.
- /**mnt** : Temporary mount directory for mounting file system.
- /**opt** : Optional is abbreviated as opt. Contains third party application software. Viz., Java, etc.
- /**proc** : A virtual and pseudo file-system which contains information about running process with a particular Process-id aka pid.
- /**root** : This is the home directory of root user and should never be confused with '/'
- /**run** : This directory is the only clean solution for early-runtime-dir problem.
- /**sbin** : Contains binary executable programs, required by System Administrator, for Maintenance. Viz., iptables, fdisk, ifconfig, swapon, reboot, etc.
- /**srv** : Service is abbreviated as 'srv'. This directory contains server specific and service related files.
- /**sys** : Modern Linux distributions include a /sys directory as a virtual filesystem, which stores and allows modification of the devices connected to the system.
- /**tmp** :System's Temporary Directory, Accessible by users and root. Stores temporary files for user and system, till next boot.
- /**usr** : Contains executable binaries, documentation, source code, libraries for second level program.
- /**var** : Stands for variable. The contents of this file is expected to grow. This directory contains log, lock, spool, mail and temp files

COMMAND FOR NAVIGATING THE LINUX FILE SYSTEMS

The following commands are used to navigate the system

Pwd:- To find the name of the working directory
cd:- To move around the file system use the cd command.
ls:- To list files and directories
file:- to identifies the file type (binary, text, etc)
cat:- To Displays a filename
cp:- To Copies one file/directory to the specified location
mv:- To Moves the location of, or renames a file/directory
mkdir:- To Creates the specified directory
rmdir:- To Removes a directory
whereis:- To Shows the location of a file

PIPING AND REDIRECTION

UNIX commands and files can be used in conjunction with one another to turn simple commands into much more complex commands. When these conjunctions involve files, we call it redirection. When these conjunctions involve commands we call it piping. Piping and redirection is the means by which we may connect these streams between programs and files to direct data in interesting and useful ways.

Pipes allow us to funnel the output from one command into another where it will be used as the input. In other words, the standard output from one program becomes the standard input for another. The "more" command takes the standard input and paginates it on the standard output (the screen). This means that if a command displays more information than can be shown on one screen, the "more" program will pause after the first screen full (page) and wait for the user to press SPACE to see the next page or RETURN to see the next line.

Here is an example which will list all the files, with details (-la) in the /dev directory and pipe the output to more. The /dev directory should have dozens of files and hence ensure that more needs to paginate.

ls -la /dev | more

Notice the --More-- prompt at the bottom of the screen. Press SPACE to see the next page and keep pressing SPACE until the output is finished.
Here is another pipe example, this time using the "wc" (word count) tool.

ls -1 /dev | wc

wc counts the numbers of lines, words and characters in the standard input. If we use the -l parameter it will display only the number of lines, which is good way to see how many files are in a directory!

Redirection is similar to pipes except using files rather than another program. The standard output for a program is the screen. Using the > (greater than) symbol the output of a program can be sent to a file. Here is a directory listing of /dev again but this time redirected to a file called listing.txt

ls -la > listing.txt

There won't be anything displayed on the terminal as everything was sent to the file. We can take a look at the file using the cat command (which can be piped into more) or for convenience we can just use the more command on its own:

more listing.txt

If listing.txt had already existed, it will be overwritten. But we can append to an existing file using >> like this:

ls -la /home > listing.txt
ls -la /dev >> listing.txt

The first redirection will overwrite the file listing.txt while the second will append to it. Now whatever text we type will be sent to the file atextfile.txt until we press Control-D, at which point the file will be closed and we will be returned to the command prompt. If we want to add more text to the file use the same command but with two greater than signs (>>).

INFORMATIONAL COMMANDS

- **ps**:- ps shows process status. Use this command to displays a table of all our own running programs or processes
- **w**:- Used to show who is logged on and what they are doing
- **id**:- Used to find out user and group names and numeric ID's (UID or group ID) of the current user or any other user in the server
- **free**:- Displays the total amount of free space available along with the amount of memory used and swap memory in the system, and also the buffers used by the kernel
- **clear**:- Used to clear the terminal screen
- **echo**:- Used to display line of text/string that are passed as an argument
- **more**:- Used to view the text files in the command prompt, displaying one screen at a time in case the file is large (For example log files)

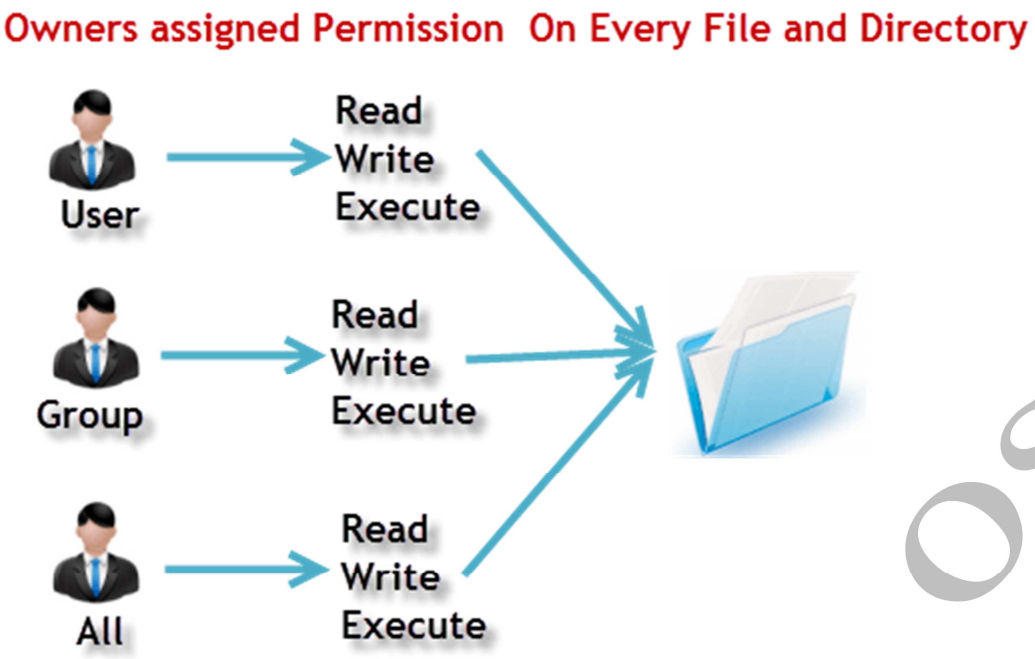
FILE PERMISSIONS

Every file and directory in our UNIX/Linux system has following 3 permissions defined for all the 3 owners (User, Group, Other).

- **Read**: This permission give we the authority to open and read a file. Read permission on a directory gives we the ability to lists its content.
- **Write**: The write permission gives we the authority to modify the contents of a file. The write permission on a directory gives we the authority to add, remove and rename files stored in the directory. Consider a scenario where we have to write permission on file but do not have write permission on the directory where the file is stored. We will be able to modify the file contents. But we will not be able to rename, move or remove the file from the directory.
- **Execute**: In Windows, an executable program usually has an extension ".exe" and which we can easily run. In Unix/Linux, we cannot run a program unless the execute permission is set. If the execute permission is not set, we might still be able to see/modify the program code(provided read & write permissions are set), but not run it.

Setting Permissions

- chmod +rwx filename to add permissions.
- chmod -rwx directoryname to remove permissions.
- chmod +x filename to allow executable permissions.
- chmod -wx filename to take out write and executable permissions.



CREATING SHELL PROGRAMS

For creating Shell programs we need to identify some terminologies

COMMENTS

When we are writing shell programs as a Systems Administrator comments are doubly important. Chances are won't be at the site forever. At some stage another Systems Administrator is going to come along and have to decipher what our shell programs are doing.

Shell programs use the # character to signify comments. Anything after a # character until the end of the line is considered a comment and is ignored by the shell.

Example

```
#!/bin/bash
# A Simple Shell Script To Get Linux Network Information
# Vivek Gite - 30/Aug/2009
echo "Current date : $(date) @ $(hostname)"
echo "Network configuration"
```

VARIABLES

A variable is declared without a $, but has a $ when invoked. Let's edit our hello-world example to use a variable for the entity being greeted, which is World.

```
#!/bin/bash
who="World"
echo Hello, $who!
```

OPERATORS

1. Arithmetic Operators

+ (Addition)	Adds values on either side of the operator
– (Subtraction)	Subtracts right hand operand from left hand operand
* (Multiplication)	Multiplies values on either side of the operator
/ (Division	Divides left hand operand by right hand operand
% (Modulus)	Divides left hand operand by right hand operand and returns remainder
(++) Increment Operator	Unary operator used to increase the value of operand by one.
(--) Decrement Operator	unary operator used to decrease the value of a operand by one

Example

```
#!/bin/sh
a=5
b=10
val=`expr $a + $b`
echo "a + b : $val"

val=`expr $a - $b`
echo "a - b : $val"

val=`expr $a \* $b`
echo "a * b : $val"
```

Relational operators

Relational operators are those operators which defines the relation between two operands. They give either true or false depending upon the relation. They are of 6 types:

'==' Operator : Double equal to operator compares the two operands. Its returns true is they are equal otherwise returns false.

'!=' Operator : Not Equal to operator return true if the two operands are not equal otherwise it returns false.

'<' Operator : Less than operator returns true if first operand is lees than second operand otherwse returns false.

'<=' Operator : Less than or equal to operator returns true if first operand is less than or equal to second operand otherwise returns false

'>' Operator : Greater than operator return true if the first operand is greater than the second operand otherwise return false.

'>=' Operator : Greater than or equal to operator returns true if first operand is greater than or equal to second operand otherwise returns false

Example

```
#!/bin/bash
#reading data from the user
read -p 'Enter a : ' a
read -p 'Enter b : ' b
```

```
if(( $a!=$b ))
then
     echo a is not equal to b.
else
     echo a is equal to b.
fi
if(( $a<=$b ))
then
     echo a is less than or equal to b.
else
     echo a is not less than or equal to b.
fi
```

LOGICAL OPERATORS

They are also known as boolean operators. These are used to perform logical operations. They are of 3 types:

1. **Logical AND (&&)** : This is a binary operator, which returns true if both the operands are true otherwise returns false.
2. **Logical OR (||)** : This is a binary operator, which returns true is either of the operand is true or both the operands are true and returns false if none of then is false.
3. **Not Equal to (!)** : This is a uninary operator which returns true if the operand is false and returns false if the operand is true.

Example

```
#!/bin/bash
#reading data from the user
read -p 'Enter a : ' a
read -p 'Enter b : ' b

if(($a == "true" & $b == "true" ))
then
     echo Both are true.
else
     echo Both are not true.
fi
if( ! $a == "true" ))
then
     echo "a" was intially false.
else
     echo "a" was intially true.
fi
```

SINGLE QUOTES

Use single quote when we want to literally print everything inside the single quote. Even the special variables such as $HOSTNAME will be print as $HOSTNAME instead of printing the name of the Linux host.

```
$ echo 'Hostname=$HOSTNAME ;
Current User=`whoami` ;
Message=\$ is USD'
Hostname=$HOSTNAME ;
Current User=`whoami` ;
Message=\$ is USD
```

DOUBLE QUOTES

Use double quotes when we want to display the real meaning of special variables.

```
$ echo "Hostname=$HOSTNAME ;
Current User=`whoami` ;
Message=\$ is USD"
Hostname=dev-db ;
Current User=ramesh ;
Message=$ is USD
```

Double quotes will remove the special meaning of all characters except the following:

$ Parameter Substitution.
` Backquotes
\$ Literal Dollar Sign.
\´ Literal Backquote.
\" Embedded Doublequote.
\\ Embedded Backslashes.

CONDITIONAL COMMANDS

There are total 5 conditional statements which can be used in bash programming

1. if statement
2. if-else statement
3. if..elif..else..fi statement (Else If ladder)
4. if..then..else..if..then..fi..fi..(Nested if)
5. switch statement

Their description with syntax is as follows

If statement

This block will process if specified condition is true.

Syntax:

```
if [ expression ]
then
   statement
fi
```

if-else statement

If specified condition is not true in if part then else part will be execute.

Syntax

```
if [ expression ]
then
   statement1
else
   statement2
fi
```

if..elif..else..fi statement (Else If ladder)

To use multiple conditions in one if-else block, then elif keyword is used in shell. If expression1 is true then it executes statement 1 and 2, and this process continues. If none of the condition is true then it processes else part.

Syntax

```
if [ expression1 ]
then
   statement1
   statement2
   .
elif [ expression2 ]
then
   statement3
   statement4
   .
else
   statement5
fi
```

if..then..else..if..then..fi..fi..(Nested if)

Nested if-else block can be used when, one condition is satisfies then it again checks another condition. In the syntax, if expression1 is false then it processes else part, and again expression2 will be check.

Syntax:

```
if [ expression1 ]
then
   statement1
   statement2
   .
else
   if [ expression2 ]
   then
      statement3
      .
   fi
fi
```

Switch Statement

The case statement works as a switch statement if specified value match with the pattern then it will execute a block of that particular pattern When a match is found all of the associated statements until the double semicolon (;;) is executed. A case will be terminated when the last command is executed. If there is no match, the exit status of the case is zero.

Syntax:

```
case  in
   Pattern 1) Statement 1;;
   Pattern n) Statement n;;
esac
```

Example 1	Example 2

#Initializing two variables
a=10
b=20
 	
#Check whether they are equal
if [$a == $b]
then
echo "a is equal to b"
fi
 	
#Check whether they are not equal
if [$a != $b]
then
echo "a is not equal to b"
fi
	#Initializing two variables
	a=20
	b=20
	
	if [$a == $b]
	then
	#If they are equal then print this
	echo "a is equal to b"
	else
	#else print this
	echo "a is not equal to b"
	fi

Example

```
CARS="bmw"
#Pass the variable in string
case "$CARS" in
     #case 1
     "mercedes")   echo   "Headquarters   -   Affalterbach,
Germany" ;;

     #case 2
     "audi") echo "Headquarters - Ingolstadt, Germany" ;;

     #case 3
     "bmw")  echo  "Headquarters  -  Chennai,  Tamil  Nadu,
India" ;;
esac
```

ITERATIVE COMMANDS

There are total 2 looping statements which can be used in bash programming

1. while statement
2. for statement

While Statement

Here command is evaluated and based on the result loop will executed, if command raise to false then loop will be terminated

Syntax

```
while command
do
   Statement to be executed
done
```

For Statement

The for loop operate on lists of items. It repeats a set of commands for every item in a list. Here var is the name of a variable and word1 to wordN are sequences of characters separated by spaces (words). Each time the for loop executes, the value of the variable var is set to the next word in the list of words, word1 to wordN.

Syntax

```
for var in word1 word2 ...wordn
do
   Statement to be executed
done
```

Example 1	**Example 1**
#Start of for loop for a in 1 2 3 4 5 6 7 8 9 10 do # if a is equal to 5 break the loop if [$a == 5] then break fi # Print the value echo "Iteration no $a" done	for a in 1 2 3 4 5 6 7 8 9 10 do # if a = 5 then continue the loop and # don't move to line 8 if [$a == 5] then continue fi echo "Iteration no $a" done

Example

```
a=0
# -lt is less than operator

#Iterate the loop until a less than 10
while [ $a -lt 10 ]
do
     # Print the values
```

```
        echo $a

        # increment the value
        a=`expr $a + 1`
    done
```

BREAK – CONTINUE

Break command is used to exit out of current loop completely before the actual ending of loop. It comes handy when we don't have prior knowledge of how long will the loop last like when we user's input is required.

Example

```
    #!/bin/bash
    #breaking a loop
    num=1
    while [ $num –lt 10 ]
    do
    if [ $num –eq 5 ]
    then
    break
    fi
    done
    echo "Loop is complete"
```

Continue command is used in script to skip current iteration of loop & continue to next iteration of the loop.

Example

```
    #!/bin/bash
    # using continue command
    for i in 1 2 3 4 5 6 7 8 9
    do
    if [ $i –eq 5 ]
    then
    echo "skipping number 5"
    continue
    fi
    echo "I is equal to $i"
    done
```

EVALUATING EXPRESSIONS

The expr is a command line Unix utility which evaluates an expression and outputs the corresponding value. expr evaluates integer or string expressions, including pattern matching regular expressions. Most of the challenge posed in writing expressions is preventing the invoking command line shell from acting on characters intended for expr to process. The operators available for integers: addition, subtraction, multiplication, division and modulus for strings: find regular expression, find a set of

characters in a string; in some versions: find substring, length of string for either: comparison (equal, not equal, less than, etc.)

Options Tag	Description
--help	Display a help message and exit.
--version	Display version information and exit.

Examples:

To perform addition of two numbers: $ expr 3 + 5 output: 8	To increament variable : $ y=10 $ y=`expr $y + 1` $ echo $y output:11	To perform multiplication $ expr 5 * 3 output:15
To find the index/position of character in a string a=hello b=`expr index $a l` echo $b output: 3 (as letter l is at position 3.)	To find substring of string: a=hello b=`expr substr $a 2 3` echo $b output: ell	To find length of string a=hello b=`expr length $a` echo $b output: 5

The bc command is used for command line calculator. It is similar to basic calculator by using which we can do basic mathematical calculations. Arithmetic operations are the most basic in any kind of programming language. Linux or Unix operating system provides the bc command and expr command for doing arithmetic calculations. We can use these commands in bash or shell script also for evaluating arithmetic expressions.

Syntax:

bc [-hlwsqv] [long-options] [file ...]

Options:

-h, {- -help } : Print the usage and exit
-i, {- -interactive } : Force interactive mode
-l, {- -mathlib } : Define the standard math library
-w, {- -warn } : Give warnings for extensions to POSIX bc
-s, {- -standard } : Process exactly the POSIX bc language
-q, {- -quiet } : Do not print the normal GNU bc welcome
-v, {- -version } : Print the version number and copyright and quit

Examples

Input : $ echo "12+5" | bc Output : 17 Input : $ echo "10^2" | bc Output : 100	Input: $ x=`echo "12+5" | bc` $ echo $x Output:17	Input: $ echo "var=10;var" | bc Output: 10
Input: $ echo "10>5" | bc Output: 1	Input: $ x=`echo "var=500;var%=7;var" | bc`	Input: $ echo "var=10;++var" | bc Output: 11

Input: $ echo "1==2" | bc Output: 0	$ echo $x Output:3	

STRINGS

Finding length of a string x=geeks len=`expr length $x` echo $len	Finding substring of a string x=geeks sub=`expr substr $x 2 3` echo $sub	Matching number of characters in two strings $ expr geeks : geek

GREP

The grep filter searches a file for a particular pattern of characters, and displays all lines that contain that pattern. The pattern that is searched in the file is referred to as the regular expression (grep stands for globally search for regular expression and print out).

Syntax:

grep [options] pattern [files]

Options Description

-c : This prints only a count of the lines that match a pattern
-h : Display the matched lines, but do not display the filenames.
-i : Ignores, case for matching
-l : Displays list of a filenames only.
-n : Display the matched lines and their line numbers.
-v : This prints out all the lines that do not matches the pattern
-e exp : Specifies expression with this option. Can use multiple times.
-f file : Takes patterns from file, one per line.
-E : Treats pattern as an extended regular expression (ERE)
-w : Match whole word
-o : Print only the matched parts of a matching line,
with each such part on a separate output line.

Example

- Case insensitive search : The -i option enables to search for a string case insensitively in the give file. It matches the words like “UNIX”, “Unix”, “unix”.

  $grep -i "UNix" geekfile.txt

- Displaying the count of number of matches : We can find the number of lines that matches the given string/pattern

  $grep -c "unix" geekfile.txt

- Display the file names that matches the pattern : We can just display the files that contains the given string/pattern.

  *$grep -l "unix" **

Or *$grep -l "unix" f1.txt f2.txt f3.xt f4.txt*

- Checking for the whole words in a file : By default, grep matches the given string/pattern even if it found as a substring in a file. The -w option to grep makes it match only the whole words.
  $ grep -w "unix" geekfile.txt

- Displaying only the matched pattern: By default, grep displays the entire line which has the matched string. We can make the grep to display only the matched string by using the -o option.
  $ grep -o "unix" geekfile.txt

ARRAYS

An array is a data structure consist multiple elements based on key pair basis. Each array element is accessible via a key index number. An Array is a data structure that stores a list (collection) of objects (elements) that are accessible using zero-based index. In Linux shells, arrays are not bound to a specific data type; there is no array of data type integer, and array of data type float. An array can contain an integer value in one element, and a string value in the element next to it.

Defining Array Values

The difference between an array variable and a scalar variable can be explained as follows. Suppose we are trying to represent the names of various students as a set of variables. Each of the individual variables is a scalar. Here array_name is the name of the array, index is the index of the item in the array that we want to set, and value is the value we want to set for that item.

```
NAME[0]="Zara"
NAME[1]="Qadir"
NAME[2]="Mahnaz"
NAME[3]="Ayan"
NAME[4]="Daisy"
```

If we are using the bash shell, here is the syntax of array initialization –

```
array_name=(value1 ... valuen)
```

Accessing Array Values

```
#!/bin/sh
NAME[0]="Zara"
NAME[1]="Qadir"
NAME[2]="Mahnaz"
NAME[3]="Ayan"
NAME[4]="Daisy"
echo "First Index: ${NAME[0]}"
echo "Second Index: ${NAME[1]}"
```

Example

```
# !/bin/bash
# To declare static Array
arr=(1 12 31 4 5)
i=0
# Loop upto size of array
# starting from index, i=0
while [ $i -lt ${#arr[@]} ]
do
        # To print index, ith
        # element
        echo ${arr[$i]}

        # Increment the i = i + 1
        i=`expr $i + 1`
done
```

```
# !/bin/bash
# To declare static Array
arr=(1 2 3 4 5)

# loops iterate through a
# set of values until the
# list (arr) is exhausted
for i in "${arr[@]}"
do
        # access each element
        # as $i
        echo $i
done
```

Module III
OPERATING SYSTEM

CPU SCHEDULING

CPU scheduling is a process which allows one process to use the CPU while the execution of another process is on hold(in waiting state) due to unavailability of any resource like I/O etc, thereby making full use of CPU. The aim of CPU scheduling is to make the system efficient, fast and fair.

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

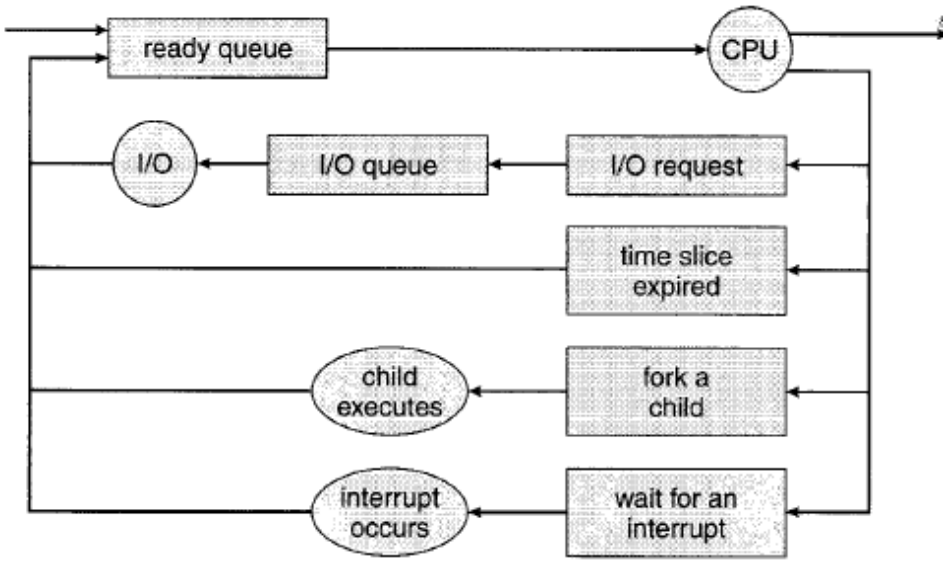


Figure: Queuing-diagram representation of process scheduling

Dispatcher

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

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

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

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

1. When a process switches from the running state to the waiting state(for I/O request or invocation of wait for the termination of one of the child processes).
2. When a process switches from the running state to the ready state (for example, when an interrupt occurs).
3. When a process switches from the waiting state to the ready state(for example, completion of I/O).
4. When a process terminates.

In circumstances 1 and 4, there is no choice in terms of scheduling. A new process(if one exists in the ready queue) must be selected for execution. There is a choice, however in circumstances 2 and 3.

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

Non-Preemptive Scheduling

Under non-preemptive scheduling, once the CPU has been allocated to a process, the process keeps the CPU until it releases the CPU either by terminating or by switching to the waiting state. This scheduling method is used by the Microsoft Windows 3.1 and by the Apple Macintosh operating systems It is the only method that can be used on certain hardware platforms, because It does not require the special hardware(for example: a timer) needed for preemptive scheduling.

When a process switches from the running state to the waiting state (for example, I/O request, or invocation of wait for the termination of one of the child processes) or when a process terminates is called non-preemptive scheduling.

Preemptive Scheduling

In this type of Scheduling, the tasks are usually assigned with priorities. At times it is necessary to run a certain task that has a

higher priority before another task although it is running. Therefore, the running task is interrupted for some time and resumed later when the priority task has finished its execution.

When a process switches from the running state to the ready state (for example, when an interrupt occurs) or when a process switches from the waiting state to the ready state (for example, completion of I/O) is called preemptive scheduling.

Preemptive	Non-preemptive
When a process switches from running state to ready state (interrupts)	When a process switches from running state to waiting state(I/O request)
When a process switches waiting state to ready state (completion of I/O)	When a process terminate

SCHEDULING CRITERIA

- **CPU Utilization**:- we wants to keep the CPU as busy as possible. Its range from 0 to 100 percent. In real time system it is 40 percent (lightly loaded system) to 90 percent (for a heavily used system)
- **Throughput**:- the number of process completed per time unit. For a long process this rate may be one process per hour, for short transactions it may be 10 processes per second.
- **Turn around time**:- It is the interval from the time of submission of a process to the time of completion. It is the sum of the periods spent waiting to get into memory, waiting in the ready queue, executing on the CPU, and doing I/O.
- **Waiting time**:- the amount of time that a process spends waiting in the ready queue. It is the sum of the periods spends waiting in the ready queue.
- **Response time**:- It is the time between the submission of the process and will produce the first output (to start responding).

Note:-We want to maximize CPU utilization and throughput and to minimize turnaround time, waiting time and response time.

SCHEDULING ALGORITHMS

A Process Scheduler schedules different processes to be assigned to the CPU based on particular scheduling algorithms. Scheduling of processes is done to finish the work on time. Below are different times with respect to a process.

Arrival Time:- Time at which the process arrives in the ready queue.
Completion Time:- Time at which process completes its execution.
Burst Time:-Time required by a process for CPU execution.
Turn Around Time:-Time Difference between completion time and arrival time.

Turn Around Time = Completion Time - Arrival Time

Waiting Time(W.T):-Time Difference between turn around time and burst time.

Waiting Time = Turn Around Time - Burst Time

There are six popular process scheduling algorithms

1. First Come First Serve(FCFS) Scheduling
2. Shortest-Job-First(SJF) Scheduling
3. Priority Scheduling
4. Round Robin(RR) Scheduling
5. Multilevel Queue Scheduling
6. Multilevel Feedback Queue Scheduling

FIRST COME FIRST SERVE(FCFS) SCHEDULING

The simplest CPU-scheduling algorithm is the first-come, first-served (FCFS) scheduling algorithm. The process that requests the CPU first is allocated the CPU first. The implementation of the FCFS policy is easily managed with a FIFO queue.

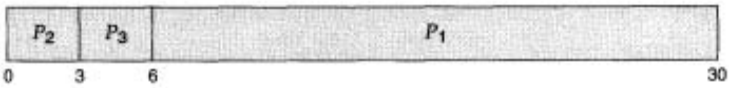
Consider the following set of processes that arrive at time 0, with the length of the CPU-burst time given in milliseconds:

Process: P1, P2, P3
Burst Time: 24, 3, 3

If the processes arrive in the order PI, P2, P3, and are served in FCFS order, we get the result shown in the following Gantt chart:

The waiting time is 0 milliseconds for process P1, 24 milliseconds for process P2, and 27 milliseconds for process P3. Thus, the average waiting time is (0 + 24 +27) / 3 = 17 milliseconds. If the processes arrive in the order P2, P3, P1, however, the results will be as shown in the following Gantt chart:

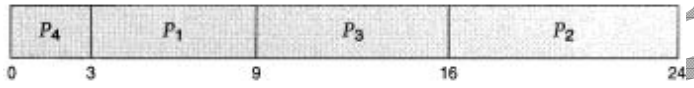
The average waiting time is now (6 + 0 + 3)/3 That is 3 milliseconds.



SHORTEST-JOB-FIRST(SJF) SCHEDULING

This algorithm associates with each process the length of the latter's next CPU burst. When the CPU is available, it is assigned to the process that has the smallest next CPU burst. If two processes have the same length next CPU burst, FCFS scheduling is used.

Consider the following set of processes, with the length of the CPU-burst time given in milliseconds:
Process: P1, P2, P3, P4
Burst Times: 6, 7, 8, 3
Using SJF scheduling, we get the following Gantt chart:

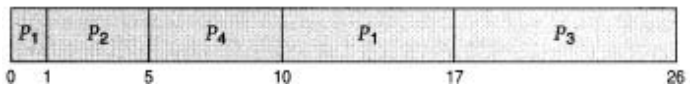


The waiting time is 3 milliseconds for process P1, 16 milliseconds for process P2, 9 milliseconds for process P3, and 0 milliseconds for process P4. Thus, the average waiting time is (3 + 16 + 9 + 0)/4 = 7 milliseconds. If we were using the FCFS scheduling scheme, then the average waiting time would be 10.25 milliseconds. The SJF algorithm may be either preemptive or non-preemptive.

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

Shortest-Remaining-Time-First scheduling (SRTF)

The Preemptive SJF scheduling is called Shortest-Remaining-Time-First scheduling. Consider the following four processes, with the length of the CPU-burst time given in milliseconds:
Process: P1, P2, P3, P4
Arrival Time: 0, 1, 2, 3
Burst Time: 8, 4, 9, 5



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

Process P1 is started at time 0, since it is the only process in the queue. Process P2 arrives at time 1. The remaining time for process P1 (7 milliseconds) is larger than the time required by process P2 (4 milliseconds), so process P1 is preempted, and process P2 is scheduled.

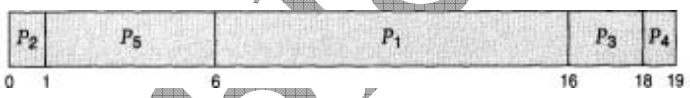
The average waiting time for this example is ((10 - 1) + (1 - 1) + (17 - 2) + (5 - 3))/4 = 26/4 = 6.5 milliseconds. A non-preemptive SJF scheduling would result in an average waiting time of 7.75 milliseconds

PRIORITY SCHEDULING

A priority is associated with each process, and the CPU is allocated to the process with the highest priority. Equal-priority processes are scheduled in FCFS order. An SJF algorithm is simply a priority algorithm where the priority (p) is the inverse of the (predicted) next CPU burst. The larger the CPU burst, the lower the priority, and vice versa. Priorities are generally some fixed range of numbers, such as 0 to 7, or 0 to 4,095. We use low numbers to represent high priority.

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

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



Using priority scheduling, we would schedule these processes according to the above Gantt chart:
The average waiting time is 8.2 milliseconds.

A major problem with priority-scheduling algorithms is indefinite blocking (or starvation). A process that is ready to run but lacking the CPU can be considered blocked-waiting for the CPU. A priority-scheduling algorithm can leave some low-priority processes waiting indefinitely for the CPU.

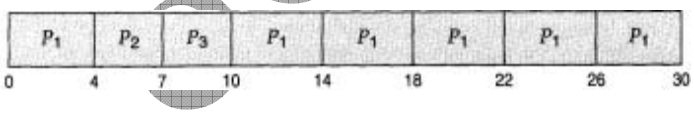
A solution to the problem of indefinite blockage of low-priority processes is aging. Aging is a technique of gradually increasing the priority of processes that wait in the system for a long time. For example, if priorities range from 127 (low) to 0 (high), we could decrement the priority of a waiting process by 1 every 15 minutes. Eventually, even a process with an initial priority of 127 would have the highest priority in the system and would be executed. In fact, it would take no more than 32 hours for a priority 127 process to age to priority 0 processes.

ROUND ROBIN(RR) SCHEDULING

The round-robin (RR) scheduling algorithm is designed especially for timesharing systems. It is similar to FCFS scheduling, but preemption is added to switch between processes. A small unit of time, called a time quantum (or time slice), is defined. A time quantum is generally from 10 to 100 milliseconds. The ready queue is treated as a circular queue.

To implement RR scheduling, we keep the ready queue as a FIFO queue of processes. New processes are added to the tail of the ready queue. The CPU scheduler picks the first process from the ready queue, sets a timer to interrupt after 1 time quantum, and dispatches the process. Consider the following set of processes that arrive at time 0, with the length of the CPU-burst time given in milliseconds:
Process: P1, P2, P3
Burst Time: 24, 3, 3

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



The average waiting time is 17/3 = 5.66 milliseconds. In the RR scheduling algorithm, no process is allocated the CPU for more than 1 time quantum in a row. If a process' CPU burst exceeds 1 time quantum, that process is preempted and is put back in the ready queue. The RR scheduling algorithm is preemptive.

MULTILEVEL QUEUE SCHEDULING

A multilevel queue-scheduling algorithm partitions the ready queue into several separate queues. The processes are permanently assigned to one queue, generally based on some property of the process, such as memory size, process priority, or process type. Each queue has its own scheduling algorithm. For example: The foreground queue might be scheduled by an RR algorithm, while the background

queue is scheduled by an FCFS algorithm. An example of a multilevel queue-scheduling algorithm with five queues:

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

MULTILEVEL FEEDBACK QUEUE SCHEDULING

Multilevel feedback queue scheduling, however, allows a process to move between queues. The idea is to separate processes with different CPU-burst characteristics. If a process uses too much CPU time, it will be moved to a lower-priority queue. This scheme leaves I/O-bound and interactive processes in the higher-priority queues. Similarly, a process that waits too long in a lower priority queue may be moved to a higher-priority queue. This form of aging prevents starvation.

A multilevel feedback queue scheduler is defined by the following parameters:

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

Multiple Processor Scheduling

If several identical processors are available, then load sharing can occur. It would be possible to provide a separate queue for each processor. In this case, one processor could be idle, with an empty queue, while another processor was very busy. To prevent this situation, we use a common ready queue. All processes go into one queue and are scheduled onto any available processor. In such a scheme, one of two scheduling approaches may be used. In one approach, each processor is self-scheduling.

Real-Time Scheduling

This scheduling facility needed to support real-time computing within a general-purpose computer system. Real-time computing is divided into two types.

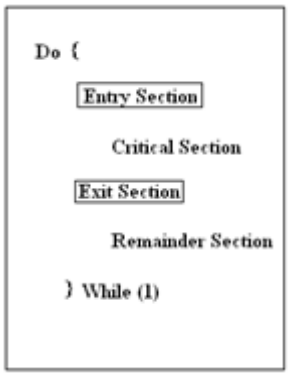
The **Hard real-time** systems are required to complete a critical task within a guaranteed amount of time. Generally, a process is submitted along with a statement of the amount of time in which it needs to complete or perform I/O. The scheduler then either admits the process, guaranteeing that the process will complete on time, or rejects the request as impossible. This is known as resource reservation.

The **Soft real-time** computing is less restrictive. It requires that critical processes receive priority over less fortunate ones. Although adding soft real-time functionality to a time-sharing system may cause an unfair allocation of resources and may result in longer delays, or even starvation, for some processes, it is at least possible to achieve.

PROCESS SYNCHRONIZATION

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

CRITICAL SECTION PROBLEM



Consider a system consisting of n processes {P0, P1, ..., Pn-1}. Each process has a segment of code, called a critical section, in which the process may be changing common variables, updating a table, writing a file, and so on. When one process is executing in its critical section, no other process is to be allowed to execute in its critical section. Thus, the execution of critical sections by the processes is mutually exclusive in time. Each process must request permission to enter its critical section. The section of code implementing this request is the entry section. The critical section may be followed by an exit section. The remaining code is the remainder section. A picture (right side) showing general structure of a process Pi.

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

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

SYNCHRONIZATION HARDWARE

Many systems provide hardware support for critical section code. The critical section problem could be solved easily in a single-processor environment if we could disallow interrupts to occur while a shared variable or resource is being modified. In this manner, we could be sure that the current sequence of instructions would be allowed to execute in order without pre-emption. Unfortunately, this solution is not feasible in a multiprocessor environment.

Disabling interrupt on a multiprocessor environment can be time consuming as the message is passed to all the processors. This message transmission lag, delays entry of threads into critical section and the system efficiency decreases.

Mutex Locks

To generalize the solution(s) expressed above, each process when entering their critical section must set some sort of lock, to prevent other processes from entering their critical sections simultaneously, and must release the lock when exiting their critical section, to allow other processes to proceed. Obviously it must be possible to attain the lock only when no other process has already set a lock. Specific implementations of this general procedure can get quite complicated, and may include hardware solutions as outlined in this section.

As the synchronization hardware solution is not easy to implement for everyone, a strict software approach called Mutex Locks was introduced. In this approach, in the entry section of code, a LOCK

is acquired over the critical resources modified and used inside critical section, and in the exit section that LOCK is released. As the resource is locked while a process executes its critical section hence no other process can access it.

- A lock is one form of hardware support for mutual exclusion.
- If a shared resource has a locked hardware lock, it is already in use by another process.
- If it is not locked, a process may freely
- Lock it for itself;
- Use it;
- Unlock it when it finishes.

Problem: Race conditions

Test-and-Set:

Another approach is for hardware to provide certain atomic operations. These operations are guaranteed to operate as a single instruction, without interruption. One such operation is the "Test and Set", which simultaneously sets a Boolean lock variable and returns its previous value

Is a hardware implementation for testing for the lock and resetting it to locked.

If test shows unlocked, the process may proceed.

- Acts as an atomic operation
- Permits
- Busy waits
- Spinning
- Spinlocks

Atomic Swap:

One simple solution to the critical section problem is to simply prevent a process from being interrupted while in their critical section, which is the approach taken by non preemptive kernels. Unfortunately this does not work well in multiprocessor environments, due to the difficulties in disabling and the re-enabling interrupts on all processors. There is also a question as to how this approach affects timing if the clock interrupt is disabled.

Performs three operations atomically

- Swap current lock value with temp locked lock
- Examine new value of temp lock
- If locked, repeat
- If unlocked, proceed into critical section/region

Utilizes a temporary variable

CLASSICAL PROBLEMS OF SYNCHRONIZATION

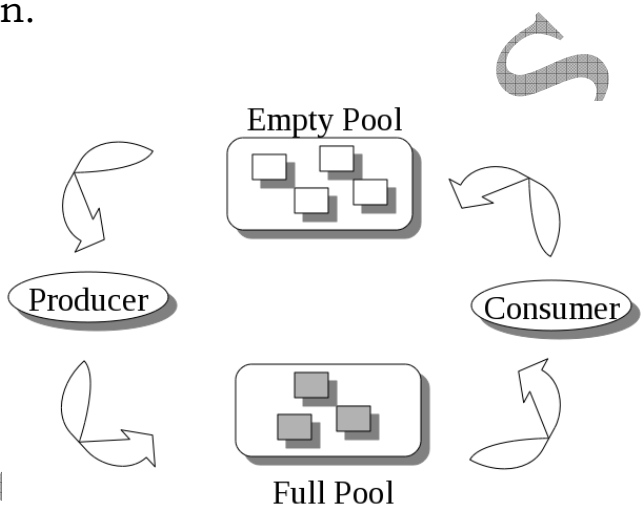
Some synchronization problems, such as the Shared Buffer Problem are fairly simple to solve using reliable synchronization facilities such as Dijkstra's Semaphore. Other problems that programmers must solve are more complex, in fact, some of them are just down right tricky to solve correctly such that there is never a problem with Starvation, Deadlock or Data Inconsistency. Fortunately, there are a lot of really smart people who have already tackled these problems and we have known solutions to them. These problems are ones that frequently arise, so we discuss them and their solutions in fairly generic terms and refer to them as the Classic Synchronization Problems. One of the biggest challenges that the programmer must solve is to correctly identify their problem as an instance of one of the classic problems. It may require thinking about the problem or framing it in a less than obvious way, so that a known solution may be used. The advantage to using a using known solution is assurance that it is correct. Some instances of deadlock, can occur only in rare conditions and are difficult to detect. So for any programming task using synchronization, it is best to reference a known solution to one of the classic problems.

1. Bounded-Buffer Problem
2. Readers and Writers Problem
3. Dining-Philosophers Problem

Bounded Buffer (Producers and Consumers)

In this problem, two processes, one called the producer and the other called the consumer, run concurrently and share a common buffer. The producer generates items that it must pass to the consumer, who is to consume them. The producer passes items to the consumer through the buffer. However, the producer must be certain that it does not deposit an item into the buffer when the buffer is full, and the consumer must not extract an item from an empty buffer. The two processes also must not access the buffer at the same time, for if the consumer tries to extract an item from the slot into which the producer is depositing an item, the consumer might get only part of the item. Any solution to this problem must ensure none of the above three events occur.

A practical example of this problem is electronic mail. The process you use to send the mail must not insert the letter into a full mailbox (otherwise the recipient will never see it); similarly, the recipient must not read a letter from an empty mailbox (or he might obtain something meaningless but that looks like a letter). Also, the sender (producer) must not deposit a letter in the mailbox at the same time the recipient extracts a letter from the mailbox; otherwise, the state of the mailbox will be uncertain.



Because the buffer has a maximum size, this problem is often called the bounded buffer problem. A (less common) variant of it is the unbounded buffer problem, which assumes the buffer is infinite. This eliminates the problem of the producer having to worry about the buffer filling up, but the other two concerns must be dealt with. Producers produce a product and consumers consume the product, but both use of one of the containers each time.

Readers and Writers Problem

In this problem, a number of concurrent processes require access to some object (such as a file.) Some processes extract information from the object and are called readers; others change or insert information in the object and are called writers. The Bernstein conditions state that many readers may access the object concurrently, but if a writer is accessing the object, no other processes (readers or writers) may access the object. There are two possible policies for doing this:

1. Readers have priority over writers; that is, unless a writer has permission to access the object, any reader requesting access to the object will get it. Note this may result in a writer waiting indefinitely to access the object.

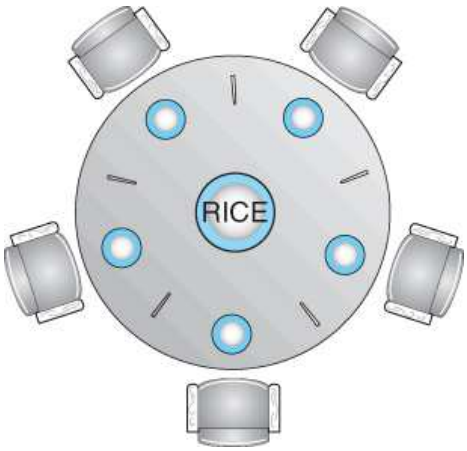
2. Writers have priority over readers; that is, when a writer wishes to access the object, only readers which have already obtained permission to access the object are allowed to complete their access; any readers that request access after the writer has done

so must wait until the writer is done. Note this may result in readers waiting indefinitely to access the object.

So there are two concerns: first, enforce the Bernstein conditions among the processes, and secondly, enforcing the appropriate policy of whether the readers or the writers have priority. A typical example of this occurs with databases, when several processes are accessing data; some will want only to read the data, others to change it. The database must implement some mechanism that solves the readers-writers problem. Writers have mutual exclusion, but multiple readers at the same time is allowed.

Dining-Philosophers Problem

In this problem, five philosophers sit around a circular table eating spaghetti and thinking. In front of each philosopher is a plate and to the left of each plate is a fork (so there are five forks, one to the right and one to the left of each philosopher's plate; see the figure). When a philosopher wishes to eat, he picks up the forks to the right and to the left of his plate. When done, he puts both forks back on the table. The problem is to ensure that no philosopher will be allowed to starve because he cannot ever pick up both forks.



There are two issues here: first, deadlock (where each philosopher picks up one fork so none can get the second) must never occur; and second, no set of philosophers should be able to act to prevent another philosopher from ever eating. A solution must prevent both.

FILE AND DATABASE SYSTEM, FILE SYSTEM, FUNCTIONS OF ORGANIZATION, ALLOCATION AND FREE SPACE MANAGEMENT.

MODULE IV
MEMORY MANAGEMENT

The memory is central to the operation of a modern computer system. Memory consists of a large array of bytes, each with its own address. The CPU fetches instructions from memory according to the value of the program counter. These instructions may cause additional loading from and storing to specific memory addresses.

A typical instruction-execution cycle, for example, first fetches an instruction from memory. The instruction is then decoded and may cause operands to be fetched from memory. After the instruction has been executed on the operands, results may be stored back in memory. The memory unit sees only a stream of memory addresses; it does not know how they are generated (by the instruction counter, indexing, indirection, literal addresses, and so on) or what they are for (instructions or data). Accordingly, we can ignore how a program generates a memory address. We are interested only in the sequence of memory addresses generated by the running program.

We begin our discussion by covering several issues that are pertinent to managing memory: basic hardware, the binding of symbolic memory addresses to actual physical addresses, and the distinction between logical and physical addresses. We conclude the section with a discussion of dynamic linking and shared libraries.

Basic Hardware

Main memory and the registers built into the processor itself are the only general-purpose storage that the CPU can access directly. There are machine instructions that take memory addresses as arguments, but none that take disk addresses. Therefore, any instructions in execution, and any data being used by the instructions, must be in one of these direct-access storage devices. If the data are not in memory, they must be moved there before the CPU can operate on them.

Registers that are built into the CPU are generally accessible within one cycle of the CPU clock. Most CPUs can decode instructions and perform simple operations on register contents at the rate of one or more operations per clock tick. The same cannot be said of main memory, which is accessed via a transaction on the memory bus. Completing a memory access may take many cycles of the CPU clock. In such cases, the processor normally needs to stall, since it does not have

the data required to complete the instruction that it is executing. This situation is intolerable because of the frequency of memory accesses. The remedy is to add fast memory between the CPU and main memory, typically on the CPU chip for fast access. To manage a cache built into the CPU, the hardware automatically speeds up memory access without any operating-system control.

Not only are we concerned with the relative speed of accessing physical memory, but we also must ensure correct operation. For proper system operation we must protect the operating system from access by user processes. On multiuser systems, we must additionally protect user processes from one another. This protection must be provided by the hardware because the operating system doesn't usually intervene between the CPU and its memory accesses (because of the resulting performance penalty). Hardware implements this production in several different ways.

We first need to make sure that each process has a separate memory space. Separate per-process memory space protects the processes from each other and is fundamental to having multiple processes loaded in memory for concurrent execution. To separate memory spaces, we need the ability to determine the range of legal addresses that the process may access and to ensure that the process can access only these legal addresses. We can provide this protection by using two registers, usually a *base register* and a *limit register*. The base register holds the smallest legal physical memory address; the limit register specifies the size of the range.

For example, if the base register holds 300040 and the limit register is 120900, then the program can legally access all addresses from 300040 through 420939 (inclusive).

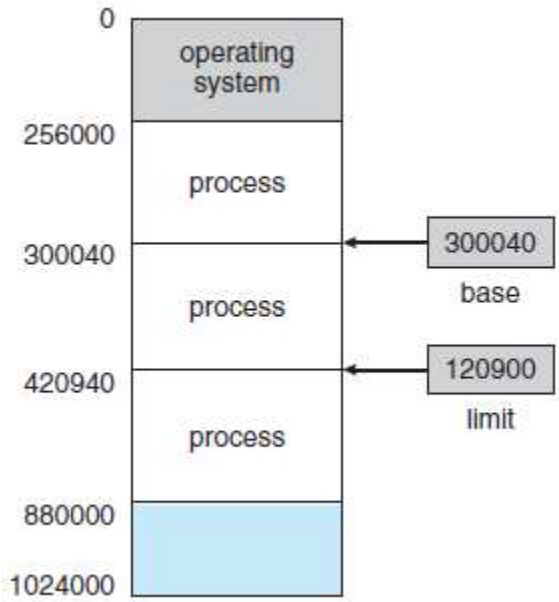


Figure 1: A base and a limit register define a logical address space.

Protection of memory space is accomplished by having the CPU hardware compare every address generated in user mode with the registers. Any attempt by a program executing in user mode to access operating-system memory or other users' memory results in a trap to the operating system, which treats the attempt as a fatal error (Figure 2). This scheme prevents a user program from (accidentally or deliberately) modifying the code or data structures of either the operating system or other users.

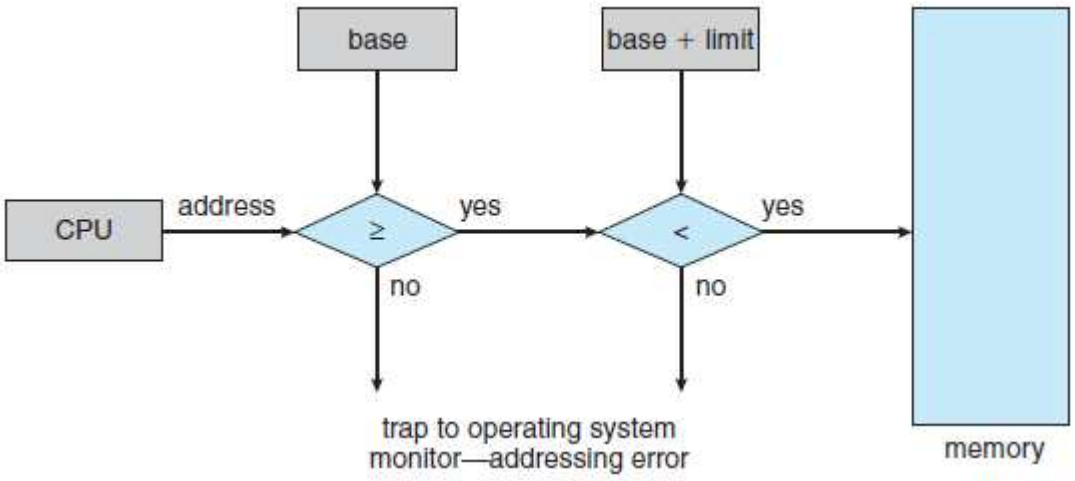


Figure 2: Hardware address protection with base and limit registers.

The base and limit registers can be loaded only by the operating system, which uses a special privileged instruction. Since privileged instructions can be executed only in kernel mode, and since only the operating system executes in kernel mode, only the operating

system can load the base and limit registers. This scheme allows the operating system to change the value of the registers but prevents user programs from changing the registers' contents.

The operating system, executing in kernel mode, is given unrestricted access to both operating-system memory and users' memory. This provision allows the operating system to load users' programs into users' memory, to dump out those programs in case of errors, to access and modify parameters of system calls, to perform I/O to and from user memory, and to provide many other services. Consider, for example, that an operating system for a multiprocessing system must execute context switches, storing the state of one process from the registers into main memory before loading the next process's context from main memory into the registers.

ADDRESS BINDING

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

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

Most systems allow a user process to reside in any part of the physical memory. Thus, although the address space of the computer may start at 00000, the first address of the user process need not be 00000. We will see later how a user program actually places a process in physical memory.

In most cases, a user program goes through several steps some of which may be optional before being executed (Figure 3). Addresses may be represented in different ways during these steps. Addresses in the source program are generally symbolic (such as the variable count). A compiler typically binds these symbolic addresses to re-locatable addresses (such as "14 bytes from the beginning of this module"). The linkage editor or loader in turn binds the re-locatable

addresses to absolute addresses (such as 74014). Each binding is a mapping from one address space to another.

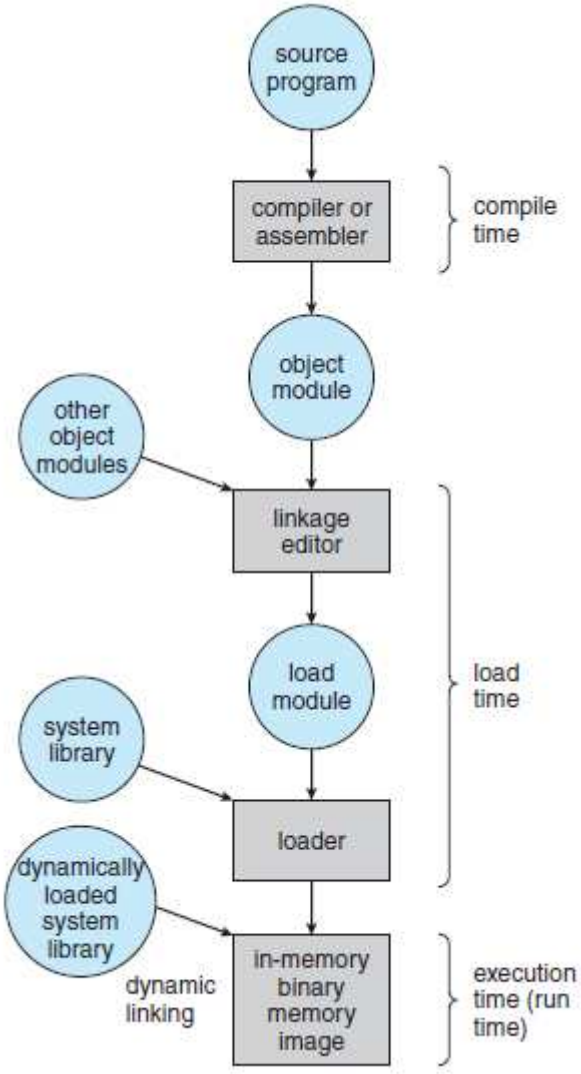


Figure 3: Multistep processing of a user program.

Classically, the binding of instructions and data to memory addresses can be done at any step along the way:

- **Compile time**. If we know at compile time where the process will reside in memory, then absolute code can be generated. For example, if we know that a user process will reside starting at location R, then the generated compiler code will start at that location and extend up from there. If, at some later time, the starting location changes, then it will be necessary to recompile this code. The MS-DOS .COM-format programs are bound at compile time.

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

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

LOGICAL VS PHYSICAL ADDRESS SPACE

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

The compile-time and load-time address-binding methods generate identical logical and physical addresses. However, the execution-time addresses binding scheme results in differing logical and physical addresses. In this case, we usually refer to the logical address as a virtual address. We use logical address and *virtual address* interchangeably in this text. The set of all logical addresses generated by a program is a *logical address space*. The set of all physical addresses corresponding to these logical addresses is a *physical address space*. Thus, in the execution-time address-binding scheme, the logical and physical address spaces differ.

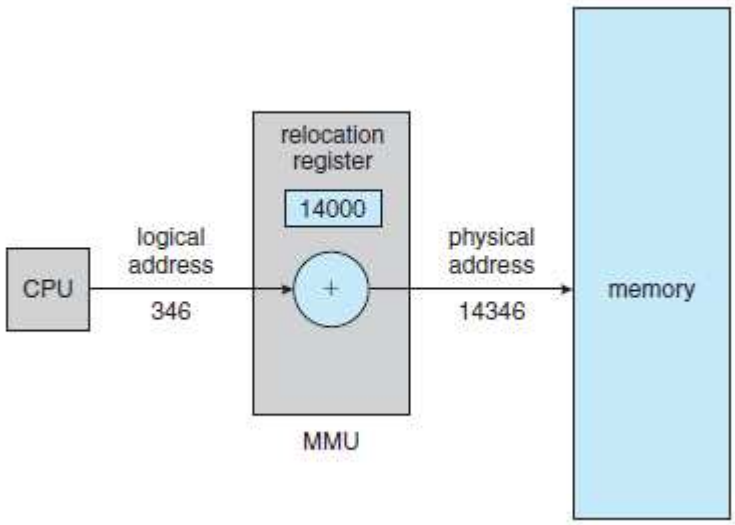


Figure 4: Dynamic relocation using a relocation register

The run-time mapping from virtual to physical addresses is done by a hardware device called the *memory-management unit* (*MMU*). We can choose from many different methods to accomplish such mapping. The base register is now called a *relocation register*. The value in the relocation register is added to every address generated by a user process at the time the address is sent to memory (see Figure 4). For example, if the base is at 14000, then an attempt by the user to address location 0 is dynamically relocated to location14000; an access to location346 is mapped to location 14346.

The user program never sees the real physical addresses. The program can create a pointer to location 346, store it in memory, manipulate it, and compare it with other addresses—all as the number 346. Only when it is used as a memory address (in an indirect load or store, perhaps) is it relocated relative to the base register. The user program deals with logical addresses. The memory-mapping hardware converts logical addresses into physical addresses. This form of execution-time binding. The final location of a referenced memory address is not determined until the reference is made. We now have two different types of addresses: logical addresses (in the range 0 to max) and physical addresses (in the range R + 0 to R + max for a base value R).

The user program generates only logical addresses and thinks that the process runs in locations 0 to max. However, these logical addresses must be mapped to physical addresses before they are used. The concept of a logical address space that is bound to a separate physical address space is central to proper memory management.

DYNAMIC LOADING

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

The advantage of dynamic loading is that a routine is loaded only when it is needed. This method is particularly useful when large amounts of code are needed to handle infrequently occurring cases, such as error routines. In this case, although the total program size may be large, the portion that is used (and hence loaded) may be much smaller.

Dynamic loading does not require special support from the operating system. It is the responsibility of the users to design their programs to take advantage of such a method. Operating systems may help the programmer, however, by providing library routines to implement dynamic loading.

DYNAMIC LINKING AND SHARED LIBRARIES

Dynamically linked libraries are system libraries that are linked to user programs when the programs are run (refer back to Figure 3). Some operating systems support only static linking, in which system libraries are treated like any other object module and are combined by the loader into the binary program image. Dynamic linking, in contrast, is similar to dynamic loading. Here, though, linking, rather than loading, is postponed until execution time. This feature is usually used with system libraries, such as language subroutine libraries. Without this facility, each program on a system must include a copy of its language library (or at least the routines referenced by the program) in the executable image. This requirement wastes both disk space and main memory.

With dynamic linking, a stub is included in the image for each library routine reference. The stub is a small piece of code that indicates how to locate the appropriate memory-resident library routine or how to load the library if the routine is not already present. When the stub is executed, it checks to see whether the needed routine is already in memory. If it is not, the program loads the routine into memory. Either way, the stub replaces itself with the address of the routine and executes the routine. Thus, the next time that particular code segment is reached, the library routine is executed directly, incurring no cost for dynamic linking. Under this scheme, all processes that use a language library execute only one copy of the library code.

This feature can be extended to library updates (such as bug fixes).A library may be replaced by a new version, and all programs that reference the library will automatically use the new version. Without dynamic linking, all such programs would need to be re-linked to gain

access to the new library. So that programs will not accidentally execute new, incompatible versions of libraries, version information is included in both the program and the library. More than one version of a library may be loaded into memory, and each program uses its version information to decide which copy of the library to use. Versions with minor changes retain the same version number, whereas versions with major changes increment the number.

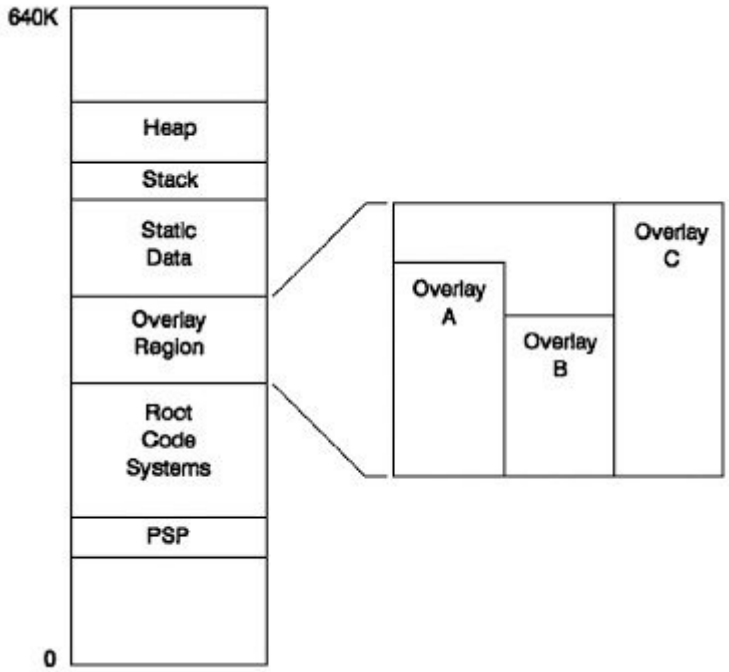
Thus, only programs that are compiled with the new library version are affected by any incompatible changes incorporated in it. Other programs linked before the new library was installed will continue using the older library. This system is also known as shared libraries. Unlike dynamic loading, dynamic linking and shared libraries generally require help from the operating system. If the processes in memory are protected from one another, then the operating system is the only entity that can check to see whether the needed routine is in another process's memory space or that can allow multiple processes to access the same memory addresses.

OVERLAYS

Overlaying means "the process of transferring a block of program code or other data into internal memory, replacing what is already stored". Overlaying is a technique that allows programs to be larger than the computer's main memory. An embedded would normally use overlays because of the limitation of physical memory which is internal memory for a system-on-chip and the lack of virtual memory facilities. Overlaying requires the programmers to split their object code to into multiple completely-independent sections, and the overlay manager that linked to the code will load the required overlay dynamically & will swap them when necessary. This technique requires the programmers to specify which overlay to load at different circumstances.

The concept of overlays is that whenever a process is running it will not use the complete program at the same time, it will use only some part of it. Then overlays concept says that whatever part we required, we load it an once the part is done, then we just unload it, means just pull it back and get the new part we required and run it. Formally, "The process of transferring a block of program code or other data into internal memory, replacing what is already stored". Sometimes it happens that compare to the size of the biggest partition,

the size of the program will be even more, then, in that case, we should go with overlays.



So overlay is a technique to run a program that is bigger than the size of the physical memory by keeping only those instructions and data that are needed at any given time. Divide the program into modules in such a way that not all modules need to be in the memory at the same time.

Advantage –

- Reduce memory requirement
- Reduce time requirement

Disadvantage –

- Overlap map must be specified by programmer
- Programmer must know memory requirement
- Overlapped module must be completely disjoint
- Programming design of overlays structure is complex and not possible in all cases

SWAPPING

A process must be in memory to be executed. A process, however, can be *swapped* temporarily out of memory to a *backing store* and then brought back into memory for continued execution (Figure 5). Swapping makes it possible for the total physical address space of all processes to exceed the real physical memory of the system, thus increasing the degree of multiprogramming in a system.

Standard Swapping

Standard swapping involves moving processes between main memory and a backing store. The backing store is commonly a fast disk. It must be large enough to accommodate copies of all memory images for all users, and it must provide direct access to these memory images. The system maintains a *ready queue* consisting of all processes whose memory images are on the backing store or in memory and are ready to run. Whenever the CPU scheduler decides to execute a process, it calls the dispatcher. The dispatcher checks to see whether the next process in the queue is in memory. If it is not, and if there is no free memory region, the dispatcher swaps out a process currently in memory and swaps in the desired process. It then reloads registers and transfers control to the selected process.

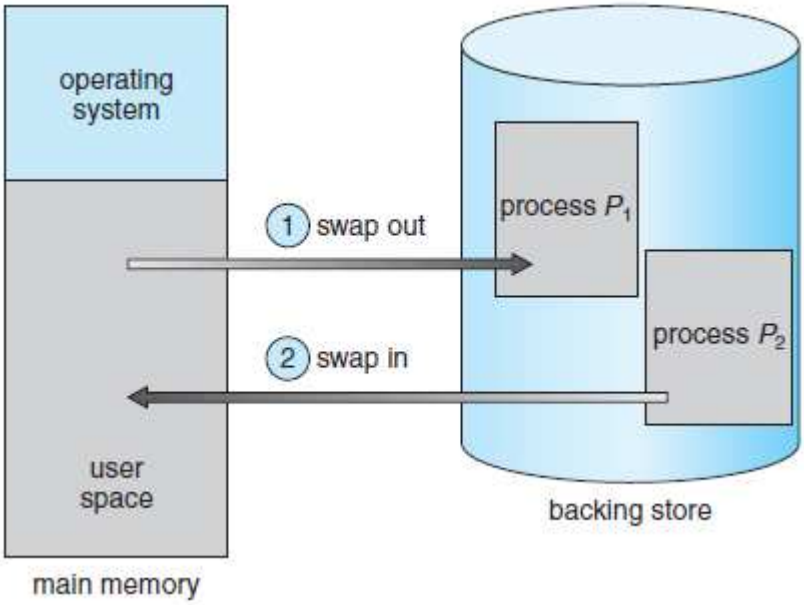


Figure 5: Swapping of two processes using a disk as a backing store.

The context-switch time in such a swapping system is fairly high. To get a idea of the context-switch time, let's assume that the user process is 100 MB in size and the backing store is a standard hard disk with a transfer rate of 50 MB per second. The actual transfer of the 100-MB process to or from main memory takes

```
100 MB/50 MB per second = 2 seconds
```

The swap time is 200 milliseconds. Since we must swap both out and in, the total swap time is about 4,000 milliseconds. Here, we are ignoring other disk performance aspects.

Notice that the major part of the swap time is transfer time. The total transfer time is directly proportional to the amount of memory swapped. If we have a computer system with 4 GB of main memory and a resident operating system taking 1 GB, the maximum size of the user process is 3 GB. However, many user processes may be much smaller than this say, 100 MB. A 100-MB process could be swapped out in 2 seconds, compared with the 60 seconds required for swapping 3 GB. Clearly, it would be useful to know exactly how much memory a user process is using, not simply how much it might be using. Then we would need to swap only what is actually used, reducing swap time. For this method to be effective, the user must keep the system informed of any changes in memory requirements. Thus, a process with dynamic memory requirements will need to issue system calls (request memory() and release memory()) to inform the operating system of its changing memory needs.

Swapping is constrained by other factors as well. If we want to swap a process, we must be sure that it is completely idle. Of particular concern is any pending I/O. A process may be waiting for an I/O operation when we want to swap that process to free up memory. However, if the I/O is asynchronously accessing the user memory for I/O buffers, then the process cannot be swapped. Assume that the I/O operation is queued because the device is busy. If we were to swap out process P1 and swap in process P2, the I/O operation might then attempt to use memory that now belongs to process P2. There are two main solutions to this problem: never swap a process with pending I/O, or execute I/O operations only into operating-system buffers. Transfers between operating-system buffers and process memory then occur only when the process is swapped in. Note that this *double buffering* itself adds overhead. We now need to copy the data again, from kernel memory to user memory, before the user process can access it.

Standard swapping is not used in modern operating systems. It requires too much swapping time and provides too little execution time to be a reasonable memory-management solution. Modified versions of swapping, however, are found on many systems, including UNIX, Linux, and Windows. In one common variation, swapping is normally disabled but will start if the amount of free memory (unused memory available for the operating system or processes to use) falls below a threshold amount. Swapping is halted when the amount of free memory increases. Another variation involves swapping portions of processes—rather than entire processes—to decrease swap time.

Typically, these modified forms of swapping work in conjunction with virtual memory.

CONTIGUOUS MEMORY ALLOCATION

In contiguous memory allocation each process is contained in a single contiguous block of memory. Memory is divided into several fixed size partitions. Each partition contains exactly one process. When a partition is free, a process is selected from the input queue and loaded into it. The free blocks of memory are known as holes. The set of holes is searched to determine which hole is best to allocate.

The main memory must accommodate both the operating system and the various user processes. We therefore need to allocate main memory in the most efficient way possible. This section explains one early method, contiguous memory allocation.

The memory is usually divided into two partitions: one for the *resident operating* system and one for the *user processes*. We can place the operating system in either low memory or high memory. The major factor affecting this decision is the location of the interrupt vector. Since the interrupt vector is often in low memory, programmers usually place the operating system in low memory as well. Thus, in this text, we discuss only the situation in which the operating system resides in low memory. The development of the other situation is similar.

We usually want several user processes to reside in memory at the same time. We therefore need to consider how to allocate available memory to the processes that are in the input queue waiting to be brought into memory. In contiguous memory allocation, each process is contained in a single section of memory that is contiguous to the section containing the next process.

Memory Allocation

Now we are ready to turn to memory allocation. One of the simplest methods for allocating memory is to divide memory into several fixed-sized partitions. Each partition may contain exactly one process.

In the fixed-partition scheme, the operating system keeps a table indicating which parts of memory are available and which are occupied. Initially, all memory is available for user processes and is considered one large block of available memory, a hole. When a process arrives and needs memory, we search for a hole large enough for this

process. If we find one, we allocate only as much memory as is needed, keeping the rest available to satisfy future requests.

As processes enter the system, they are put into an input queue. The operating system takes into account the memory requirements of each process and the amount of available memory space in determining which processes are allocated memory. When a process is allocated space, it is loaded into memory, and it can then compete for the CPU. When a process terminates, it releases its memory, which the operating system may then fill with another process from the input queue.

At any given time, we have a list of available block sizes and the input queue. The operating system can order the input queue according to a scheduling algorithm. Memory is allocated to processes until, finally, the memory requirements of the next process cannot be satisfied that is, no available block of memory (or hole) is large enough to hold that process. The operating system can then wait until a large enough block is available, or it can skip down the input queue to see whether the smaller memory requirements of some other process can be met.

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

- **First fit**. Allocate the first hole that is big enough. Searching can start either at the beginning of the set of holes or where the previous first-fit search ended. We can stop searching as soon as we find a free hole that is large enough.

- **Best fit**. Allocate the smallest hole that is big enough. We must search the entire list, unless the list is ordered by size. This strategy produces the smallest leftover hole.

- **Worst fit**. Allocate the largest hole. Again, we must search the entire list, unless it is sorted by size. This strategy produces the largest leftover hole, which may be more useful than the smaller leftover hole from a best-fit approach.

Simulations have shown that both first fit and best fit are better than worst fit in terms of decreasing time and storage utilization. Neither first fit nor best fit is clearly better than the other in terms of storage utilization, but first fit is generally faster.

Fragmentation

Both the first-fit and best-fit strategies for memory allocation suffer from *external fragmentation*. As processes are loaded and removed from memory, the free memory space is broken into little pieces. External fragmentation exists when there is enough total memory space to satisfy a request, but the available spaces are not contiguous; storage is fragmented into a large number of small holes. This fragmentation problem can be severe. In the worst case, we could have a block of free (or wasted) memory between every two processes. If all these small pieces of memory were in one big free block instead, we might be able to run several more processes.

Memory fragmentation can be internal as well as external. Consider a multiple-partition allocation scheme with a hole of 18,464 bytes. Suppose that the next process requests 18,462 bytes. If we allocate exactly the requested block, we are left with a hole of 2 bytes. The overhead to keep track of this hole will be substantially larger than the hole itself. The general approach to avoiding this problem is to break the physical memory into fixed-sized blocks and allocate memory in units based on block size. With this approach, the memory allocated to a process may be slightly larger than the requested memory. The difference between these two numbers is *internal fragmentation* memory that is internal to a partition but is not being used.

One solution to the problem of external fragmentation is *compaction*. The goal is to shuffle the memory contents so as to place all free memory together in one large block. Compaction is not always possible, however. If relocation is static and is done at assembly or load time, compaction cannot be done; compaction is possible only if relocation is dynamic and is done at execution time. If addresses are relocated dynamically, relocation requires only moving the program and data and then changing the base register to reflect the new base address. When compaction is possible, we must determine its cost. The simplest compaction algorithm is to move all processes toward one end of memory; all holes move in the other direction, producing one large hole of available memory. This scheme can be expensive.

Another possible solution to the external-fragmentation problem is to permit the logical address space of the processes to be non-contiguous, thus allowing a process to be allocated physical memory wherever the latter is available. Two complementary techniques achieve this solution: paging and segmentation.

STATIC MEMORY ALLOCATION	DYNAMIC MEMORY ALLOCATION
Memory is allocated before the execution of the program begins (During Compilation).	Memory is allocated during the execution of the program.
Variables remain permanently allocated.	Allocated only when program unit is active.
In this type of allocation Memory cannot be resized after the initial allocation.	In this type of allocation Memory can be dynamically expanded and shrunk as necessary.
Implemented using stacks.	Implemented using heap.
Faster execution than Dynamic.	Slower execution than static.
It is less efficient than Dynamic allocation strategy.	It is more efficient than Static allocation strategy.
Implementation of this type of allocation is simple.	Implementation of this type of allocation is complicated.
Memory cannot be reuse when it is no longer needed.	Memory can be freed when it is no longer needed & reuse or reallocate during execution.

Static Memory Allocation

Memory is allocated for the declared variable by the compiler. The address can be obtained by using 'address of' operator and can be assigned to a pointer. The memory is allocated during compile time. Since most of the declared variables have static memory, this kind of assigning the address of a variable to a pointer is known as static memory allocation.

The compiler allocates the required memory space for a declared variable. By using the address of operator, the reserved address is obtained and this address may be assigned to a pointer variable. Since most of the declared variable has static memory, this way of assigning pointer value to a pointer variable is known as static memory allocation. Memory is assigned during compilation time.

Dynamic Memory Allocation

Allocation of memory at the time of execution (run time) is known as dynamic memory allocation. The functions calloc() and malloc() support allocating of dynamic memory. Dynamic allocation of memory space is done by using these functions when value is returned by functions and assigned to pointer variables.

It uses functions such as malloc( ) or calloc( ) to get memory dynamically. If these functions are used to get memory dynamically and the values returned by these functions are assigned to pointer variables, such assignments are known as dynamic memory allocation. Memory is assigned during run time.

NON-CONTIGUOUS MEMORY ALLOCATION

It is preferable when dealing with large amounts of memory to use physically contiguous pages in memory both for cache related and memory access latency reasons. Unfortunately, due to external fragmentation problems with the buddy allocator, this is not always possible. The non-contiguous memory allocation assigns the separate memory blocks at the different location in memory space in a non-consecutive manner to a process requesting for memory.

SEGMENTATION

An important aspect of memory management that became unavoidable with paging is the separation of the user's view of memory and the actual physical memory. As we have already seen, the user's view of memory is not the same as the actual physical memory. The user's view is mapped onto physical memory. This mapping allows differentiation between logical memory and physical memory.

Basic Method

Segmentation is a memory-management scheme that supports this user view of memory. A logical address space is a collection of segments. Each segment has a name and a length. The addresses specify both the segment name and the offset within the segment. The user therefore specifies each address by two quantities: a segment name and an offset. (Contrast this scheme with the paging scheme, in which the user specifies only a single address, which is partitioned by the hardware into a page number and an offset, all invisible to the programmer.)

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

<segment-number, offset>

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

1. AC compiler might create separate segments for the following:
2. The code
3. Global variables
4. The heap, from which memory is allocated
5. The stacks used by each thread
6. The standard C library

Libraries that are linked in during compile time might be assigned separate segments. The loader would take all these segments and assign them segment numbers.

Segmentation Hardware

Although the user can now refer to objects in the program by a two-dimensional address, the actual physical memory is still, of course, a one-dimensional sequence of bytes. Thus, we must define an implementation to map two-dimensional user-defined addresses into one-dimensional physical addresses. This mapping is affected by a segment table. Each entry in the segment table has a segment base and a segment limit. The segment base contains the starting physical address where the segment resides in memory, whereas the segment limit specifies the length of the segment.

The use of a segment table is illustrated in Figure 3. A logical address consists of two parts: a segment number, s, and an offset into that segment, d. The segment number is used as an index to the segment table. The offset d of the logical address must be between 0 and the segment limit. If it is not, we trap to the operating system (logical addressing attempt beyond end of segment).

When an offset is legal, it is added to the segment base to produce the address in physical memory of the desired byte. The segment table is thus essentially an array of base–limit register pairs.

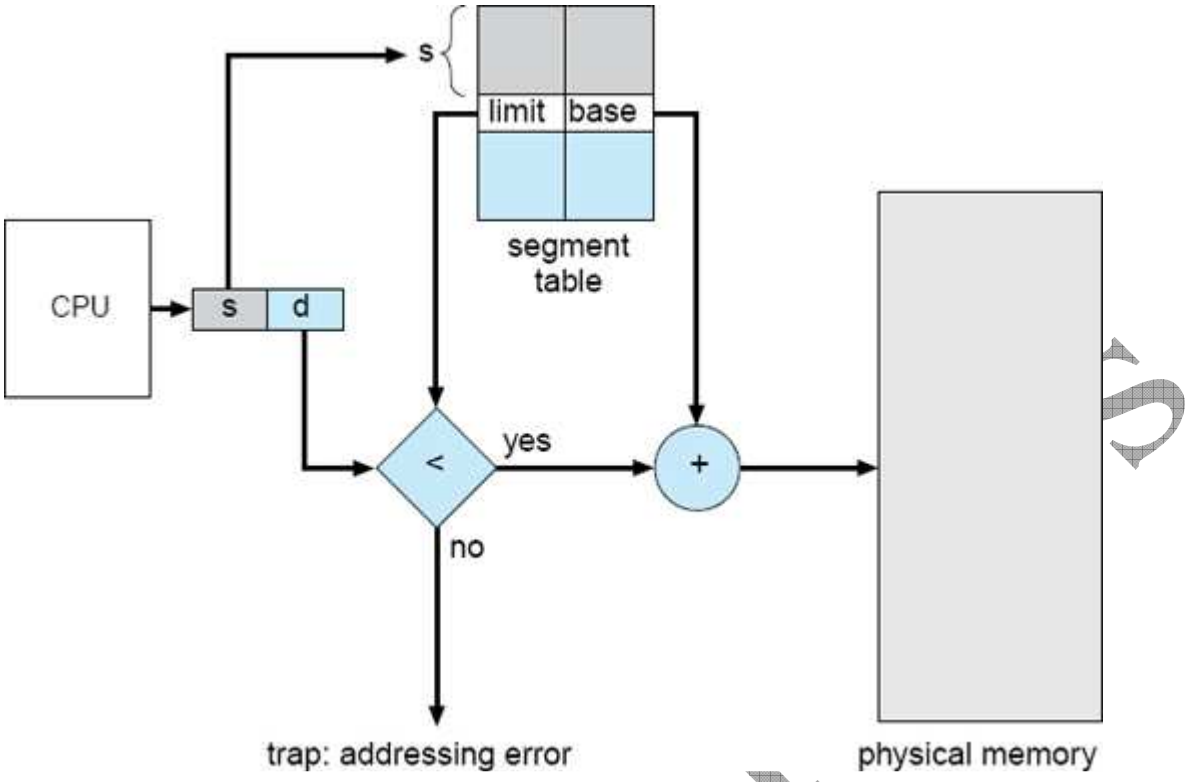


Figure 3: Segmentation hardware.

As an example, consider the situation shown in Figure 4. We have five segments numbered from 0 through 4. The segments are stored in physical memory as shown. The segment table has a separate entry for each segment, giving the beginning address of the segment in physical memory (or base) and the length of that segment (or limit). For example, segment 2 is 400 bytes long and begins at location 4300. Thus, a reference to byte 53 of segment 2 is mapped onto location 4300 + 53 = 4353. A reference to segment 3, byte 852, is mapped to 3200 (the base of segment 3) + 852 = 4052. A reference to byte 1222 of segment 0 would result in a trap to the operating system, as this segment is only 1,000 bytes long.

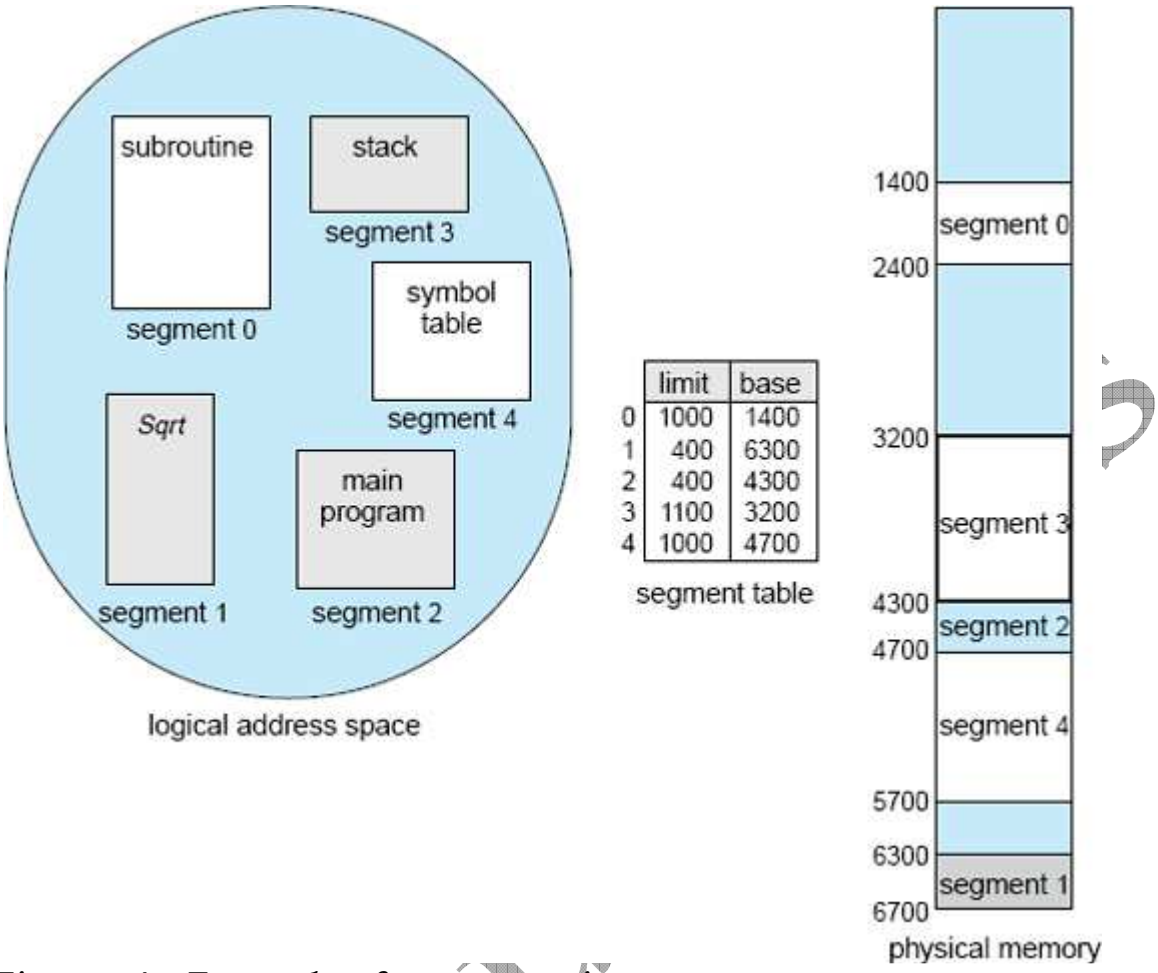


Figure 4: Example of segmentation.

PAGING

Paging is a memory-management scheme that permits the physical address space of a process to be non-contiguous. Paging avoids the considerable problem of fitting memory chunks of varying sizes onto the backing store; most memory-management schemes used before the introduction of paging suffered from this problem. The problem arises because, when some code fragments or data residing in main memory need to be swapped out, space must be found on the backing store. The backing store also has the fragmentation problems discussed in connection with main memory, except that access is much slower, so compaction is impossible. Because of its advantages over earlier methods, paging in its various forms is commonly used in most operating systems.

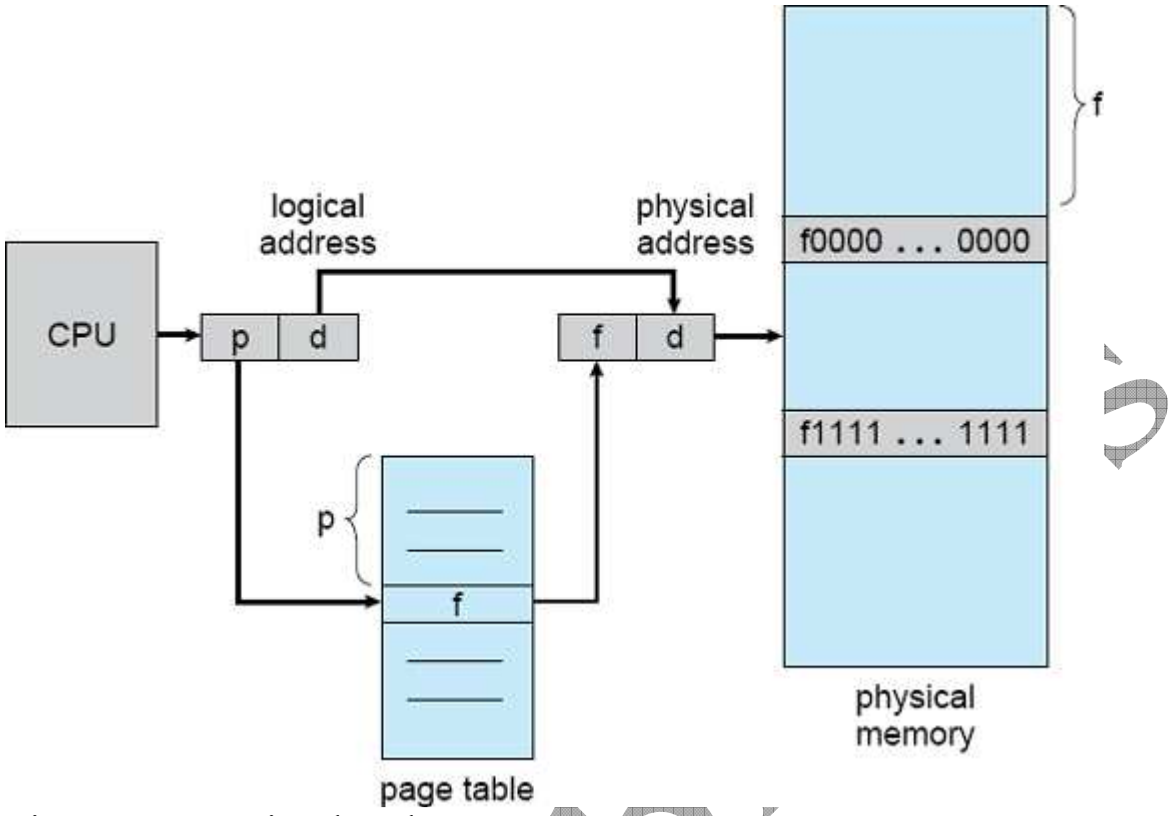


Figure 4: Paging hardware.

Traditionally, support for paging has been handled by hardware. However, recent designs have implemented paging by closely integrating the hardware and operating system, especially on 64-bit microprocessors.

Basic Method

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

The hardware support for paging is illustrated in Figure 4. Every address generated by the CPU is divided into two parts: a page number (p) and a page offset (d). The page number is used as an index into a page table. The page table contains the base address of each page in physical memory. This base address is combined with the page offset to define the physical memory address that is sent to the memory unit. The paging model of memory is shown in Figure 5.

The page size (like the frame size) is defined by the hardware. The size of a page is typically a power of 2, varying between 512 bytes and 16 MB per page, depending on the computer architecture. The selection of a power of 2 as a page size makes the translation of a logical address into a page number and page offset particularly easy. If the size of logical address space is 2m, and a page size is 2n addressing units (bytes or words), then the high-order m - n bits of a logical address designate the page number, and the n low-order bits designate the page offset. Thus, the logical address is as follows:

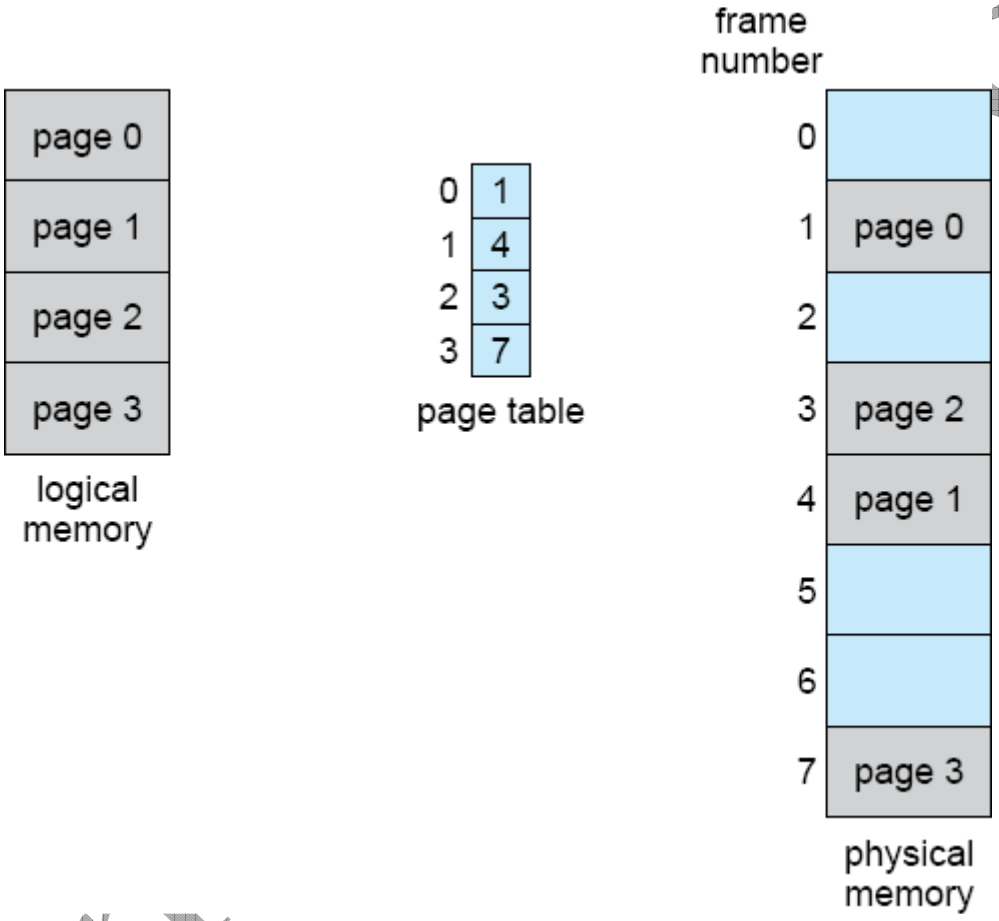


Figure 5: Paging model of logical and physical memory.

Where p is an index into the page table and d is the displacement within the page.

As a concrete (although minuscule) example, consider the memory in Figure 6. Using a page size of 4 bytes and a physical memory of 32 bytes (8 pages), we show how the user's view of memory can be mapped into physical memory. Logical address 0 is page 0, offset

0. Indexing into the page table, we find that page 0 is in frame 5. Thus, logical address 0 maps to physical address 20 (= (5 × 4) + 0). Logical address 3 (page 0, offset 3) maps to physical address 23 (= (5 × 4) + 3). Logical address 4 is page 1, offset 0; according to the page table, page 1 is mapped to frame 6. Thus, logical address 4 maps to physical address 24 (= (6 × 4) + 0). Logical address 13 maps to physical address 9.

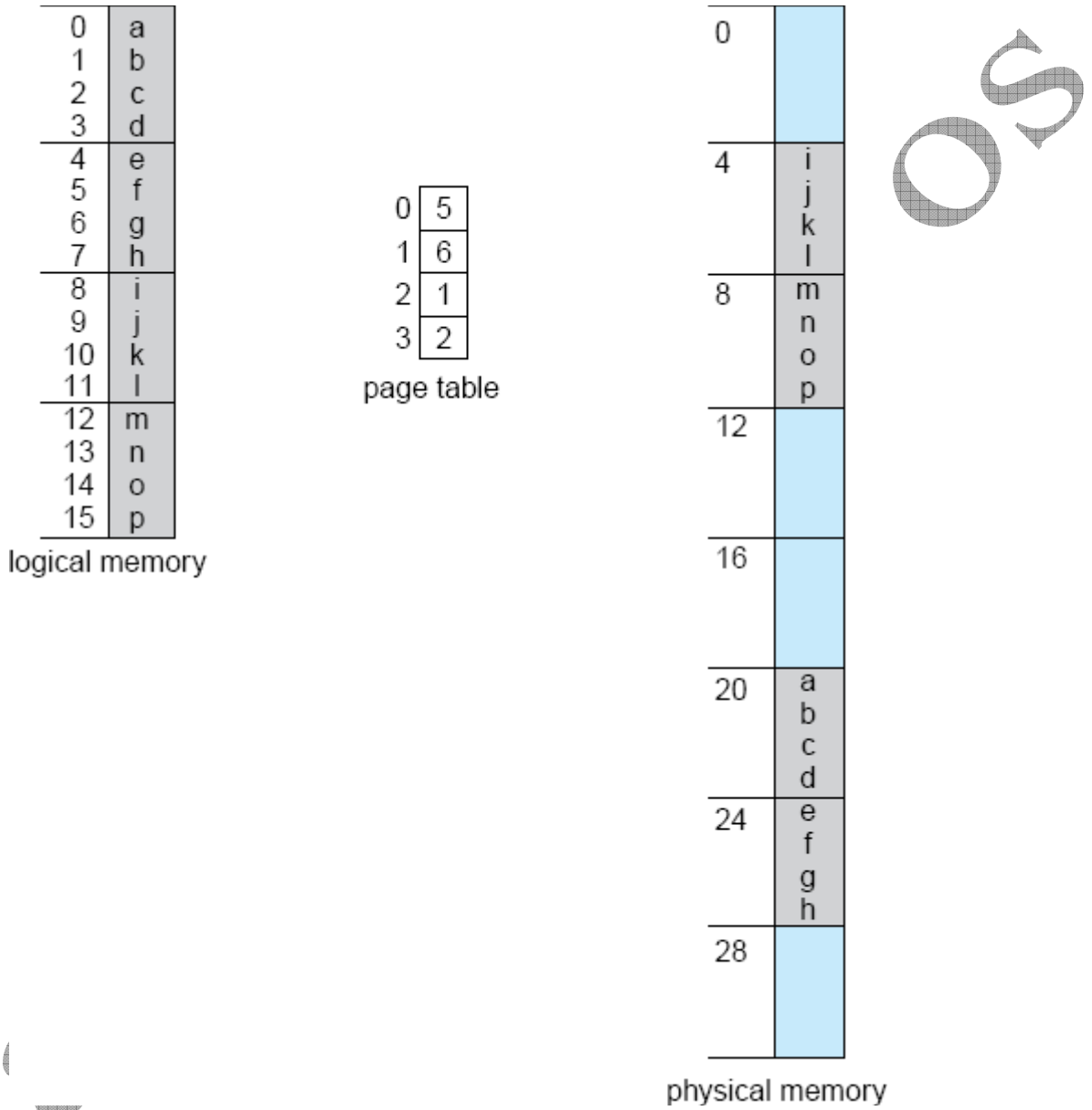


Figure 6: Paging example for a 32-byte memory with 4-byte pages.

We may have noticed that paging itself is a form of dynamic relocation. Every logical address is bound by the paging hardware to some physical address. Using paging is similar to using a table of base (or relocation) registers, one for each frame of memory.

Then we use a paging scheme, we have no external fragmentation: Any free frame can be allocated to a process that needs it. However, we may have some internal fragmentation. Notice that frames are allocated as units. If the memory requirements of a process do not happen to coincide with page boundaries, the last frame allocated may not be completely full. For example, if page size is 2,048 bytes, a process of 72,766 bytes would need 35 pages plus 1,086 bytes. It would be allocated 36 frames, resulting in an internal fragmentation of 2,048 − 1,086 = 962 bytes. In the worst case, a process would need n pages plus 1 byte. It would be allocated n + 1 frames, resulting in an internal fragmentation of almost an entire frame.

If process size is independent of page size, we expect internal fragmentation to average one-half page per process. This consideration suggests that small page sizes are desirable. However, overhead is involved in each page-table entry, and this overhead is reduced as the size of the pages increases. Also, disk I/O is more efficient when the number of data being transferred is larger. Generally, page sizes have grown over time as processes, data sets, and main memory have become larger. Today, pages typically are between 4 KB and 8 KB in size, and some systems support even larger page sizes. Some CPUs and kernels even support multiple page sizes. For instance, Solaris uses page sizes of 8 KB and 4 MB, depending on the data stored by the pages. Researchers are now developing variable on-the-fly page-size support.

Usually, each page-table entry is 4 bytes long, but that size can vary as well. A 32-bit entry can point to one of 232 physical page frames. If frame size is 4 KB, then a system with 4-byte entries can address 244bytes (or 16 TB) of physical memory.

When a process arrives in the system to be executed, its size, expressed in pages, is examined. Each page of the process needs one frame. Thus, if the process requires n pages, at least n frames must be available in memory. If n frames are available, they are allocated to this arriving process. The first page of the process is loaded into one of the allocated frames, and the frame number is put in the page table for this process. The next page is loaded into another frame, and its frame number is put into the page table, and so on (Figure 7).

An important aspect of paging is the clear separation between the user's view of memory and the actual physical memory. The user program views memory as one single space, containing only this one program. In fact, the user program is scattered throughout physical memory, which also holds other programs. The difference between the user's view of memory and the actual physical memory is reconciled by the address-translation hardware. The logical addresses are translated into physical addresses. This mapping is hidden from the user and is controlled by the operating system. Notice that the user process by definition is unable to access memory it does not own. It has no way of addressing memory outside of its page table, and the table includes only those pages that the process owns.

Since the operating system is managing physical memory, it must be aware of the allocation details of physical memory which frames are allocated, which frames are available, how many total frames there are, and so on. This information is generally kept in a data structure called a frame table. The frame table has one entry for each physical page frame, indicating whether the latter is free or allocated and, if it is allocated, to which page of which process or processes.

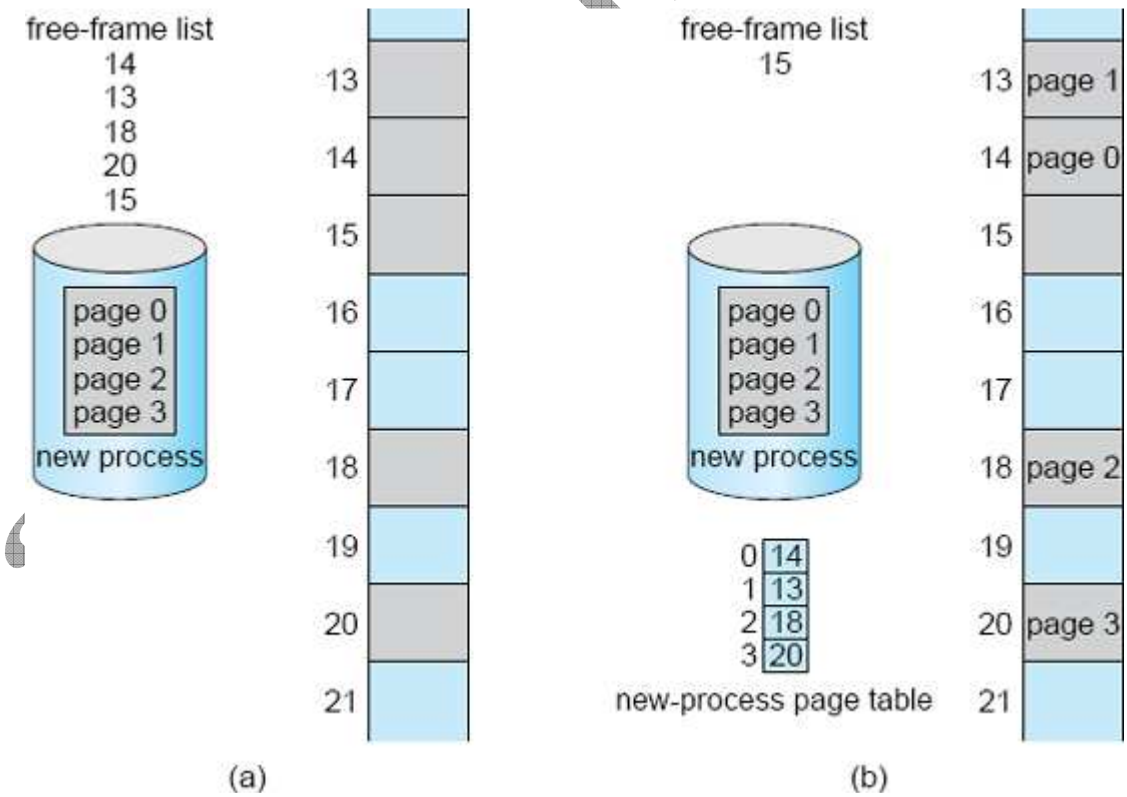


Figure 7: Free frames (a) before allocation and (b) after allocation.

In addition, the operating system must be aware that user processes operate in user space, and all logical addresses must be mapped to produce physical addresses. The operating system maintains a copy of the page table for each process, just as it maintains a copy of the instruction counter and register contents. This copy is used to translate logical addresses to physical addresses whenever the operating system must map a logical address to a physical address manually. It is also used by the CPU dispatcher to define the hardware page table when a process is to be allocated the CPU. Paging therefore increases the context-switch time.

Hardware Support

The hardware implementation of the page table can be done in several ways. In the simplest case, the page table is implemented as a set of dedicated registers. These registers should be built with very high-speed logic to make the paging-address translation efficient. Every access to memory must go through the paging map, so efficiency is a major consideration. The CPU dispatcher reloads these registers, just as it reloads the other registers. Instructions to load or modify the page-table registers are, of course, privileged, so that only the operating system can change the memory map.

The use of registers for the page table is satisfactory if the page table is reasonably small (for example, 256 entries). Most contemporary computers, however, allow the page table to be very large (for example, 1 million entries). For these machines, the use of fast registers to implement the page table is not feasible. Rather, the page table is kept in main memory, and a page-table base register (PTBR) points to the page table. Changing page tables requires changing only this one register, substantially reducing context-switch time.

The standard solution to this problem is to use a special, small, fast-lookup hardware cache, called a *translation look-aside buffer* (TLB). The TLB is associative, high-speed memory. Each entry in the TLB consists of two parts: a key (or tag) and a is. When the associative memory is presented with an item, the item is compared with all keys simultaneously. If the item is found, the corresponding value field is returned. The search is fast; the hardware, however, is expensive. Typically, the number of entries in a TLB is small, often numbering between 64 and 1,024.

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

If the page number is not in the TLB (known as a *TLB miss*), a memory reference to the page table must be made. When the frame number is obtained, we can use it to access memory (Figure 8). In addition, we add the page number and frame number to the TLB, so that they will be found quickly on the next reference. If the TLB is already full of entries, the operating system must select one for replacement. Replacement policies range from least recently used (LRU) to random. Furthermore, some TLBs allow entries to be wired down, meaning that they cannot be removed from the TLB. Typically, TLB entries for kernel code are wired down.

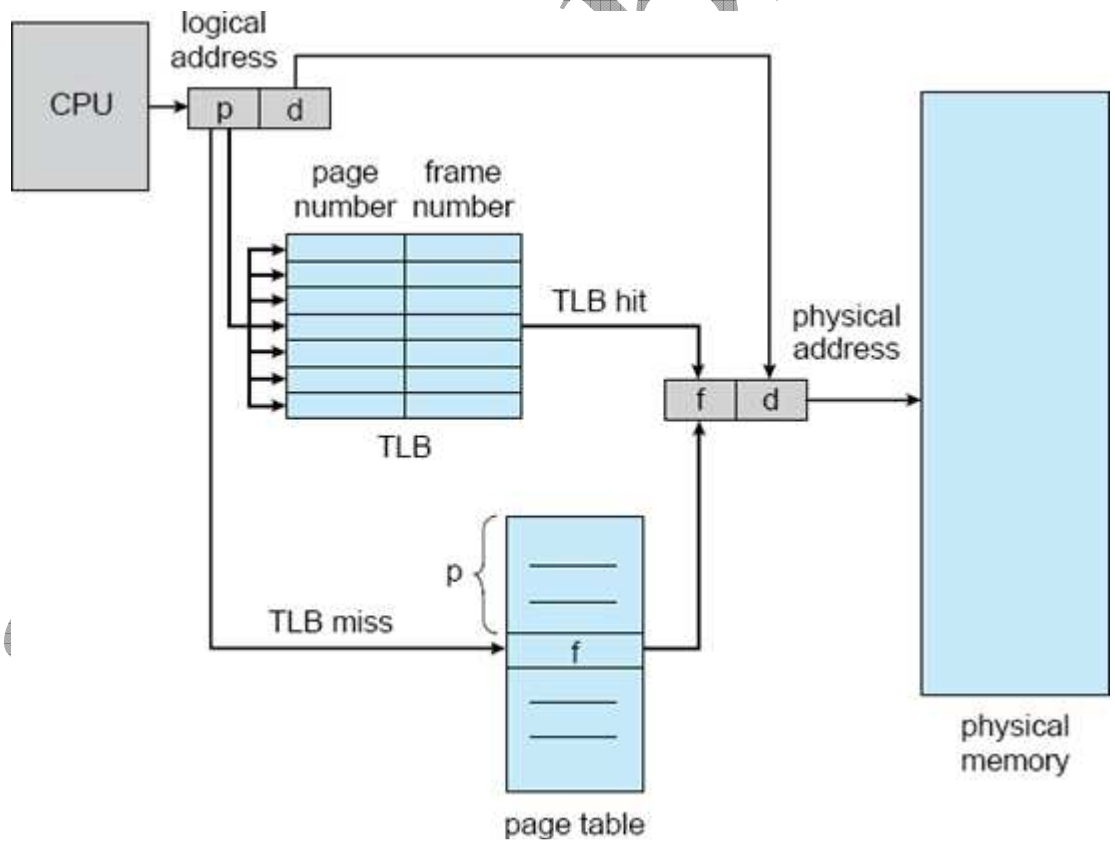


Figure 8:L Paging hardware with TLB.

The percentage of times that a particular page number is found in the TLB is called the *hit ratio*. An 80-percent hit ratio means that we find the desired page number in the TLB 80 percent of the time. If it takes 20 nanoseconds to search the TLB and 100 nanoseconds to access memory, then a mapped-memory access takes 120 nanoseconds when the page number is in the TLB. If we fail to find the page number in the TLB (20 nanoseconds), then we must first access memory for the page table and frame number (100 nanoseconds) and then access the desired byte in memory (100 nanoseconds), for a total of 220 nanoseconds. To find the effective memory-access time, we weight each case by its probability:

effective access time = 0.80 × 120 + 0.20 × 220
= 140 nanoseconds

In this example, we suffer a 40-percent slowdown in memory-access time (from 100 to 140 nanoseconds).
For a 98-percent hit ratio, we have

effective access time = 0.98 × 120 + 0.02 × 220
= 122 nanoseconds

This increased hit rate produces only a 22 percent slowdown in access time. We will further explore the impact of the hit ratio on the TLB

VIRTUAL MEMORY

Virtual memory is a technique that allows the execution of processes that are not completely in memory. One major advantage of this scheme is that programs can be larger than physical memory. Further, virtual memory abstracts main memory into an extremely large, uniform array of storage, separating logical memory as viewed by the user from physical memory. This technique frees programmers from the concerns of memory-storage limitations.

Virtual memory also allows processes to share files easily and to implement shared memory. In addition, it provides an efficient mechanism for process creation. Virtual memory is not easy to implement, however, and may substantially decrease performance if it is used carelessly. Here we discuss virtual memory in the form of demand paging and examine its complexity and cost.

The memory-management algorithms outlined are necessary because of one basic requirement: The instructions being executed must be in physical memory. The first approach to meeting this requirement is to place the entire logical address space in physical memory. Dynamic loading can help to ease this restriction, but it generally requires special precautions and extra work by the programmer.

The requirement that instructions must be in physical memory to be executed seems both necessary and reasonable; but it is also unfortunate, since it limits the size of a program to the size of physical memory. In fact, an examination of real programs shows us that, in many cases, the entire program is not needed.

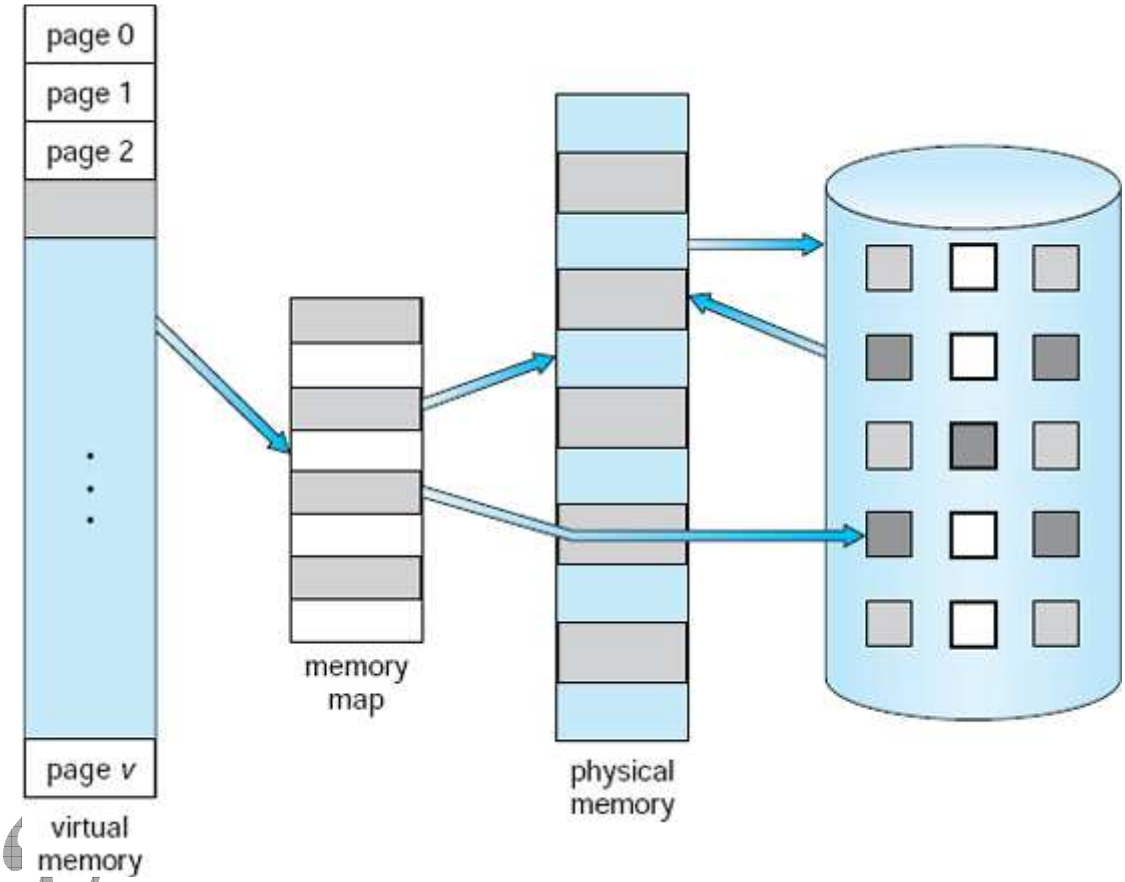


Figure 10: Diagram showing virtual memory that is larger than physical memory.

Virtual memory involves the separation of logical memory as perceived by users from physical memory. This separation allows an extremely large virtual memory to be provided for programmers when only a smaller physical memory is available (Figure 10). Virtual memory makes the task of programming much easier, because the programmer

no longer needs to worry about the amount of physical memory available; she can concentrate instead on the problem to be programmed.

The virtual address space of a process refers to the logical (or virtual) view of how a process is stored in memory. Typically, this view is that a process begins at a certain logical address say, address 0 and exists in contiguous memory, as shown in Figure 11. In fact physical memory may be organized in page frames and that the physical page frames assigned to a process may not be contiguous. It is up to the memory-management unit (MMU) to map logical pages to physical page frames in memory.

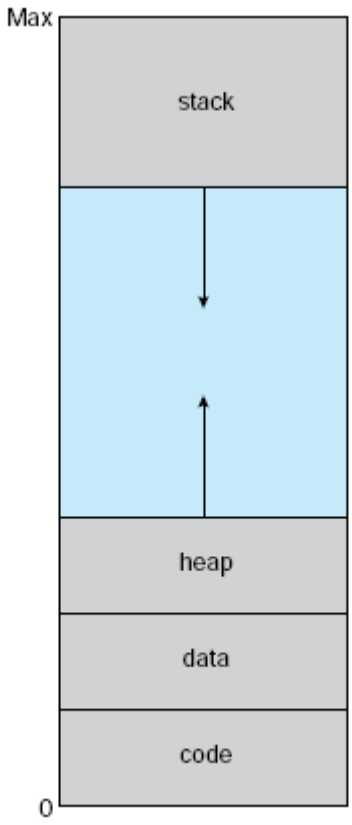


Figure 11: Virtual address space

Note in Figure 11 that we allow for the heap to grow upward in memory as it is used for dynamic memory allocation. Similarly, we allow for the stack to grow downward in memory through successive function calls. The large blank space (or hole) between the heap and the stack is part of the virtual address space but will require actual physical pages only if the heap or stack grows. Virtual address spaces that include holes are known as sparse address spaces. Using a sparse address space is beneficial because the holes can be filled as the stack or heap segments grow or if we wish to dynamically link libraries (or possibly other shared objects) during program execution.

In addition to separating logical memory from physical memory, virtual memory also allows files and memory to be shared by two or more processes through page sharing. This leads to the following benefits:

- System libraries can be shared by several processes through mapping of the shared object into a virtual address space. Although each process considers the shared libraries to be part of its virtual address space, the actual pages where the libraries reside in physical memory are shared by all the processes (Figure

12). Typically, a library is mapped read-only into the space of each process that is linked with it.

- Similarly, virtual memory enables processes to share memory. If two or more processes can communicate through the use of shared memory. Virtual memory allows one process to create a region of memory that it can share with another process. Processes sharing this region consider it part of their virtual address space, yet the actual physical pages of memory are shared, much as is illustrated in Figure 12.
- Virtual memory can allow pages to be shared during process creation with the fork() system call, thus speeding up process creation.

Implementing virtual memory through demand paging.

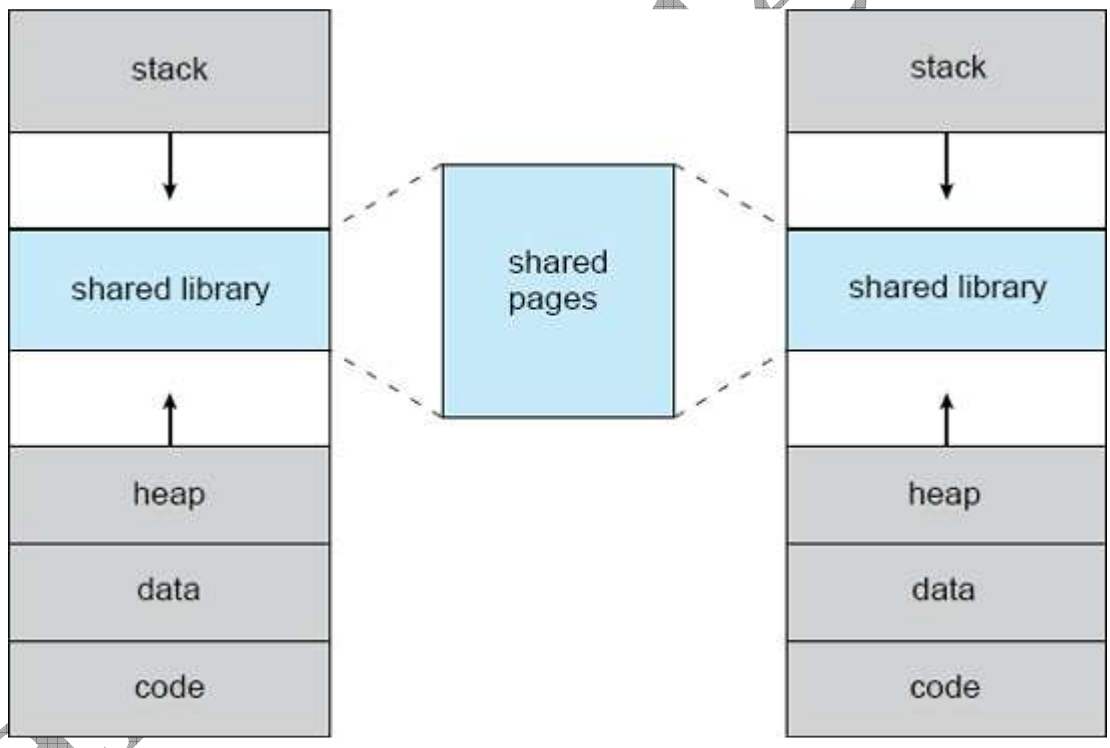


Figure 12: Shared library using virtual memory.

DEMAND PAGING

Consider a program that starts with a list of available options from which the user is to select. Loading the entire program into memory results in loading the executable code for all options, regardless of whether an option is ultimately selected by the user or not. An alternative strategy is to initially load pages only as they are

needed. This technique is known as demand paging and is commonly used in virtual memory systems. With demand-paged virtual memory, pages are only loaded when they are demanded during program execution; pages that are never accessed are thus never loaded into physical memory.

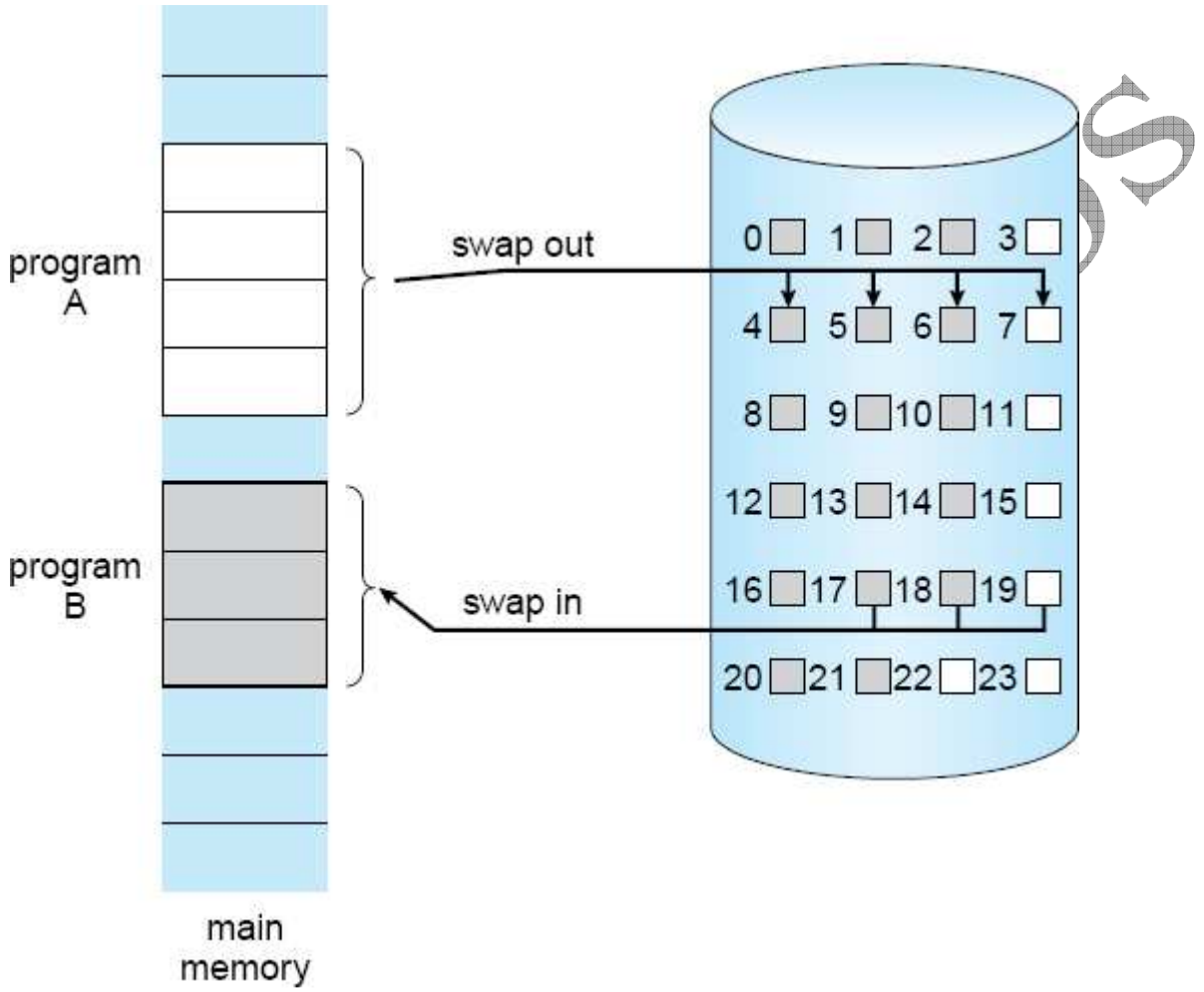


Figure 13: Transfer of a paged memory to contiguous disk space.

A demand-paging system is similar to a paging system with swapping (Figure 13) where processes reside in secondary memory (usually a disk). When we want to execute a process, we swap it into memory. Rather than swapping the entire process into memory, however, we use a lazy swapper. A lazy swapper never swaps a page into memory unless that page will be needed. Since we are now viewing a process as a sequence of pages, rather than as one large contiguous address space, use of the term swapper is technically incorrect. A swapper manipulates entire processes, whereas a pager is concerned with the individual pages of a process. We thus use pager, rather than swapper, in connection with demand paging.

Basic Concepts

When a process is to be swapped in, the pager guesses which pages will be used before the process is swapped out again. Instead of swapping in a whole process, the pager brings only those necessary pages into memory. Thus, it avoids reading into memory pages that will not be used anyway, decreasing the swap time and the amount of physical memory needed.

With this scheme, we need some form of hardware support to distinguish between the pages that are in memory and the pages that are on the disk. The valid–invalid bit scheme can be used for this purpose. This time, however, when this bit is set to "valid," the associated page is both legal and in memory. If the bit is set to "invalid," the page either is not valid (that is, not in the logical address space of the process) or is valid but is currently on the disk. The page-table entry for a page that is brought into memory is set as usual, but the page-table entry for a page that is not currently in memory is either simply marked invalid or contains the address of the page on disk. This situation is depicted in Figure 14.

Notice that marking a page invalid will have no effect if the process never attempts to access that page. Hence, if we guess right and page in all and only those pages that are actually needed, the process will run exactly as though we had brought in all pages. While the process executes and accesses pages that are memory resident, execution proceeds normally.

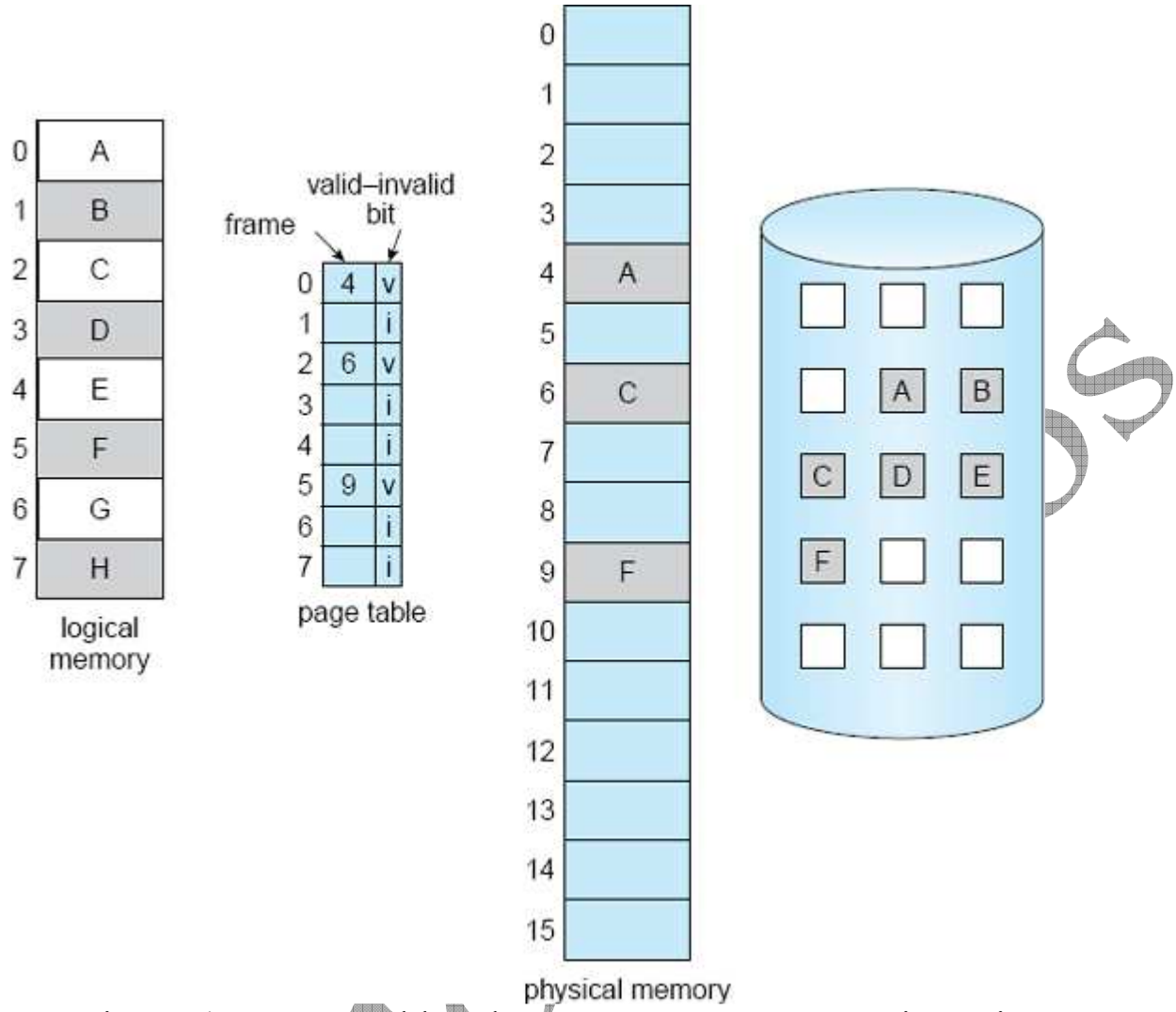


Figure 14: Page table when some pages are not in main memory.

If the process tries to access a page that was not brought into memory, then access to a page marked invalid causes a page-fault trap. The paging hardware, in translating the address through the page table, will notice that the invalid bit is set, causing a trap to the operating system.

This trap is the result of the operating system's failure to bring the desired page into memory. The procedure for handling this page fault is straightforward (Figure 15):

1. We check an internal table (usually kept with the process control block) for this process to determine whether the reference was a valid or an invalid memory access.

2. If the reference was invalid, we terminate the process. If it was valid, but we have not yet brought in that page, we now page it in.

3. We find a free frame (by taking one from the free-frame list, for example).
4. We schedule a disk operation to read the desired page into the newly allocated frame.
5. When the disk read is complete, we modify the internal table kept with the process and the page table to indicate that the page is now in memory.
6. We restart the instruction that was interrupted by the trap. The process can now access the page as though it had always been in memory.

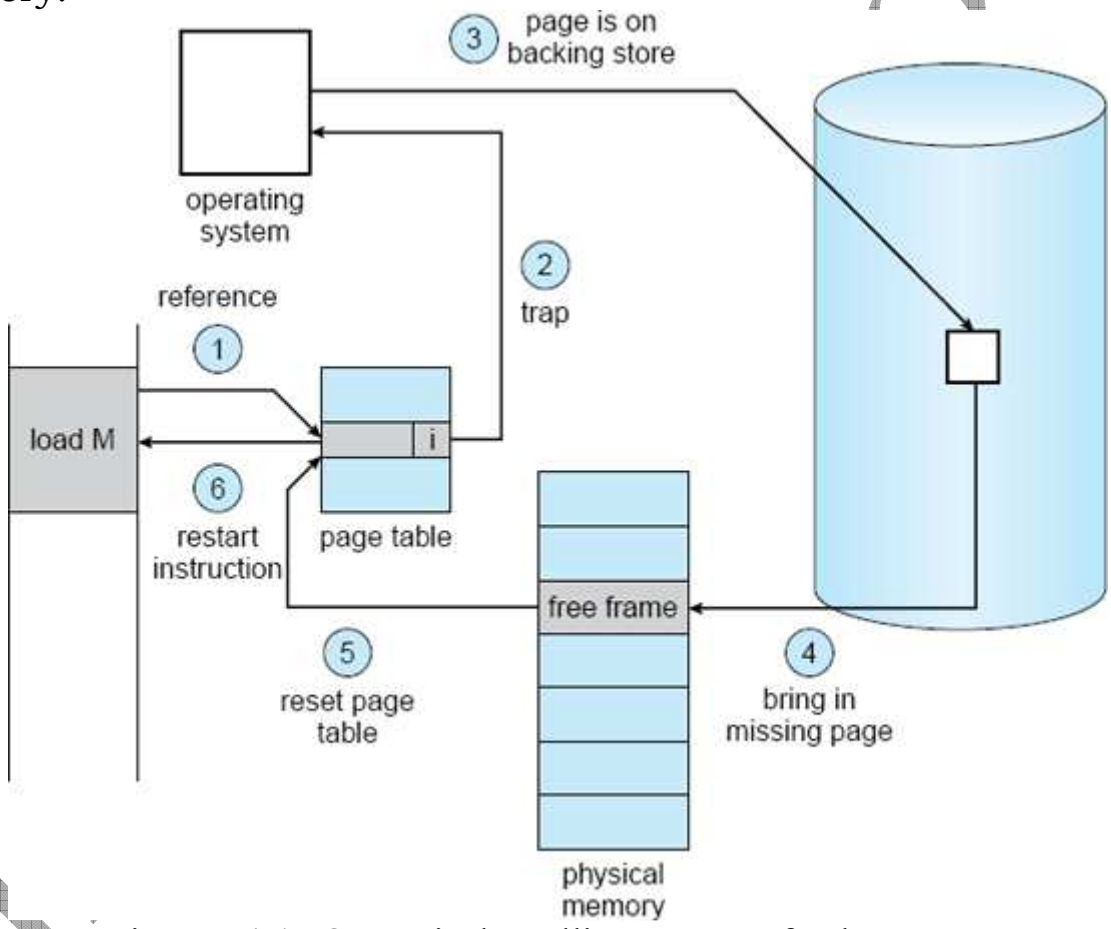


Figure 15: Steps in handling a page fault.

In the extreme case, we can start executing a process with no pages in memory. When the operating system sets the instruction pointer to the first instruction of the process, which is on a non-memory-resident page, the process immediately faults for the page. After this page is brought into memory, the process continues to execute, faulting as necessary until every page that it needs is in memory. At that point, it can execute with no more faults. This scheme is *pure demand paging*: Never bring a page into memory until it is required.

Theoretically, some programs could access several new pages of memory with each instruction execution (one page for the instruction and many for data), possibly causing multiple page faults per instruction. This situation would result in unacceptable system performance. Fortunately, analysis of running processes shows that this behaviour is exceedingly unlikely. Programs tend to have *locality of reference*, which results in reasonable performance from demand paging.

The hardware to support demand paging is the same as the hardware for paging and swapping:

- **Page table**. This table has the ability to mark an entry invalid through a valid–invalid bit or special value of protection bits.
- **Secondary memory**. This memory holds those pages that are not present in main memory. The secondary memory is usually a high-speed disk. It is known as the swap device, and the section of disk used for this purpose is known as swap space.

A crucial requirement for demand paging is the need to be able to restart any instruction after a page fault. Because we save the state (registers, condition code, and instruction counter) of the interrupted process when the page fault occurs, we must be able to restart the process in exactly the same place and state, except that the desired page is now in memory and is accessible. In most cases, this requirement is easy to meet. A page fault may occur at any memory reference. If the page fault occurs on the instruction fetch, we can restart by fetching the instruction again. If a page fault occurs while we are fetching an operand, we must fetch and decode the instruction again and then fetch the operand.

PAGE REPLACEMENT POLICIES

If a process of ten pages actually uses only half of them, then demand paging saves the I/O necessary to load the five pages that are never used. We could also increase our degree of multiprogramming by running twice as many processes. Thus, if we had forty frames, we could run eight processes, rather than the four that could run if each required ten frames (five of which were never used).

If we increase our degree of multiprogramming, we are over-allocating memory. If we run six processes, each of which is ten pages in size but actually uses only five pages, we have higher CPU utilization

and throughput, with ten frames to spare. It is possible, however, that each of these processes, for a particular data set, may suddenly try to use all ten of its pages, resulting in a need for sixty frames when only forty are available.

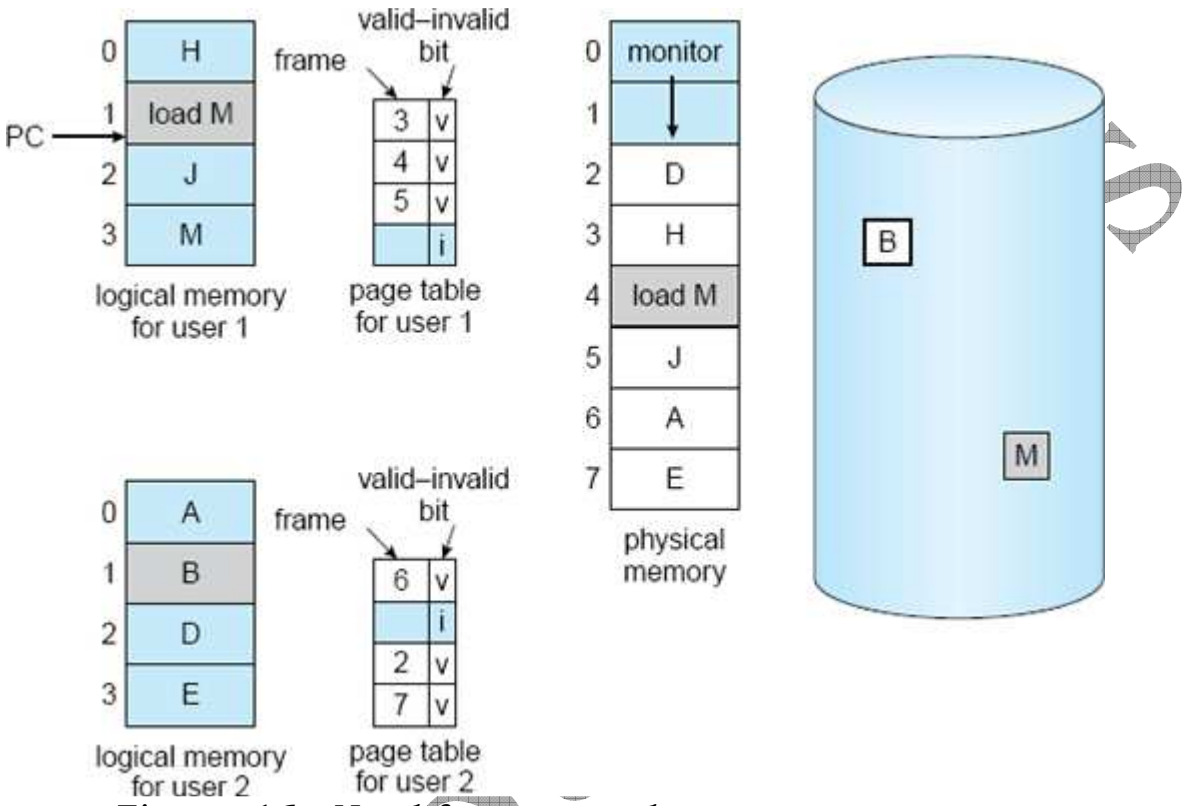


Figure 16: Need for page replacement.

Over-allocation of memory manifests itself as follows. While a user process is executing, a page fault occurs. The operating system determines where the desired page is residing on the disk but then finds that there are no free frames on the free-frame list; all memory is in use (Figure 16).

The operating system has several options at this point. It could terminate the user process. However, demand paging is the operating system's attempt to improve the computer system's utilization and throughput. Users should not be aware that their processes are running on a paged system paging should be logically transparent to the user. So this option is not the best choice.

The operating system could instead swap out a process, freeing all its frames and reducing the level of multiprogramming. This option is a good one in certain circumstances. Here, we discuss the most common solution: page replacement.

We illustrate several page-replacement algorithms. In doing so, we use the reference string for a memory with three frames.

7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

FIFO Page Replacement

The simplest page-replacement algorithm is a first-in, first-out (FIFO) algorithm. A FIFO replacement algorithm associates with each page the time when that page was brought into memory. When a page must be replaced, the oldest page is chosen. Notice that it is not strictly necessary to record the time when a page is brought in. We can create a FIFO queue to hold all pages in memory. We replace the page at the head of the queue. When a page is brought into memory, we insert it at the tail of the queue.

For our example reference string, our three frames are initially empty. The first three references (7, 0, 1) cause page faults and are brought into these empty frames. The next reference (2) replaces page 7, because page 7 was brought in first. Since 0 is the next reference and 0 is already in memory, we have no fault for this reference. The first reference to 3 results in replacement of page 0, since it is now first in line. Because of this replacement, the next reference, to 0, will fault. Page 1 is then replaced by page 0. This process continues as shown in Figure 18. Every time a fault occurs, we show which pages are in our three frames. There are 15 faults altogether.

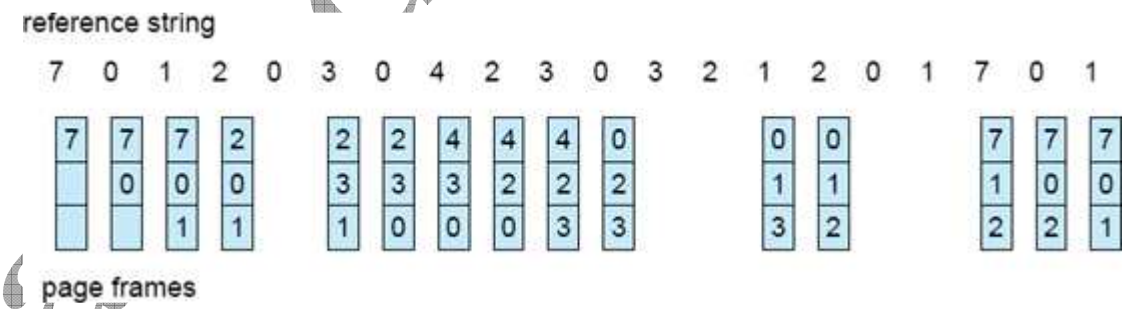


Figure 18. FIFO page-replacement algorithm.

The FIFO page-replacement algorithm is easy to understand and program. However, its performance is not always good. On the one hand, the page replaced may be an initialization module that was used a long time ago and is no longer needed. On the other hand, it could contain a heavily used variable that was initialized early and is in constant use.

Notice that, even if we select for replacement a page that is in active use, everything still works correctly. After we replace an active page with a new one, a fault occurs almost immediately to retrieve the active page. Some other page will need to be replaced to bring the active page back into memory. Thus, a bad replacement choice increases the page-fault rate and slows process execution. It does not, however, cause incorrect execution.

To illustrate the problems that are possible with a FIFO page-replacement algorithm, we consider the followisng reference string:

1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

Figure 19 shows the curve of page faults for this reference string versus the number of available frames. Notice that the number of faults for four frames (ten) is greater than the number of faults for three frames (nine)! This most unexpected result is known as Belady's anomaly: For some page-replacement algorithms, the page-fault rate may increase as the number of allocated frames increases. We would expect that giving more memory to a process would improve its performance. In some early research, investigators noticed that this assumption was not always true. Belady's anomaly was discovered as a result.

Optimal Page Replacement

One result of the discovery of Belady's anomaly was the search for an optimal page-replacement algorithm. An optimal page-replacement algorithm has the lowest page-fault rate of all algorithms and will never suffer from Belady's anomaly. Such an algorithm does exist and has been called OPT or MIN. It is simply this:

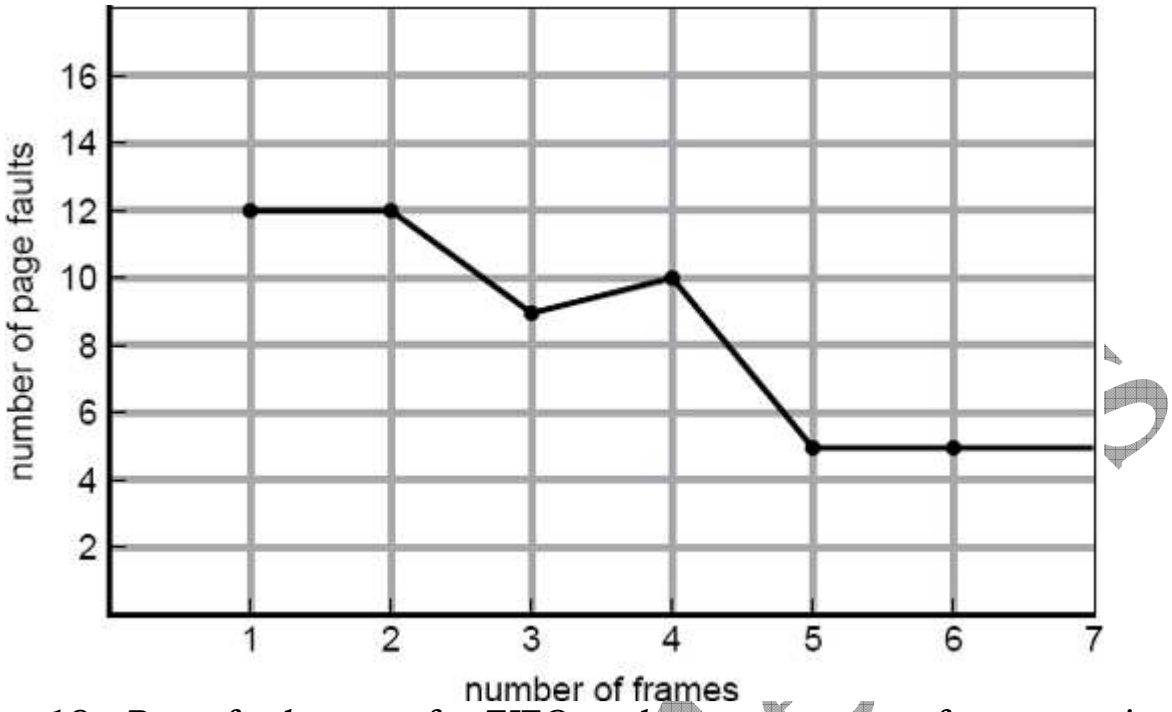


Figure 19: Page-fault curve for FIFO replacement on a reference string.

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

Use of this page-replacement algorithm guarantees the lowest possible page-fault rate for a fixed number of frames.

For example, on our sample reference string, the optimal page-replacement algorithm would yield nine page faults, as shown in Figure 20. The first three references cause faults that fill the three empty frames. The reference to page 2 replaces page 7, because 7 will not be used until reference 18, whereas page 0 will be used at 5, and page 1 at 14. The reference to page 3 replaces page 1, as page 1 will be the last of the three pages in memory to be referenced again. With only nine page faults, optimal replacement is much better than a FIFO algorithm, which resulted in fifteen faults. (If we ignore the first three, which all algorithms must suffer, then optimal replacement is twice as good as FIFO replacement.)

In fact, no replacement algorithm can process this reference string in three frames with fewer than nine faults. Unfortunately, the optimal page-replacement algorithm is difficult to implement, because it requires future knowledge of the reference string. As a result, the optimal algorithm is used mainly for comparison studies. For instance, it may be useful to know that, although a new algorithm is not optimal, it is within 12.3 percent of optimal at worst and within 4.7 percent on average.

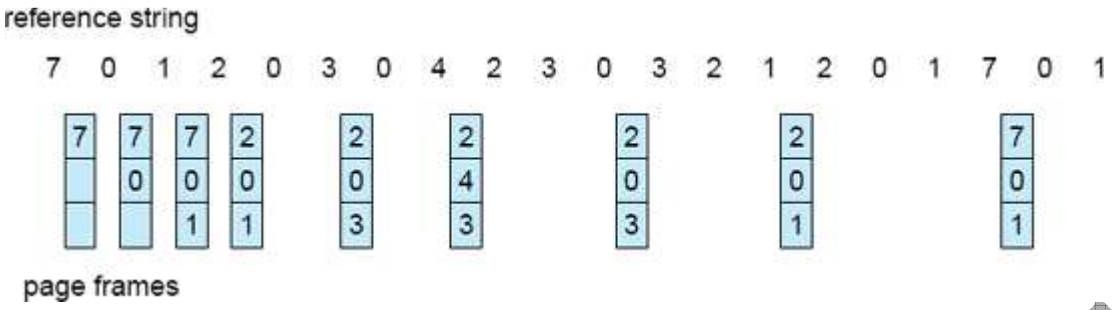


Figure 20: Optimal page-replacement algorithm.

LRU Page Replacement

If the optimal algorithm is not feasible, perhaps an approximation of the optimal algorithm is possible. The key distinction between the FIFO and OPT algorithms (other than looking backward versus forward in time) is that the FIFO algorithm uses the time when a page was brought into memory, whereas the OPT algorithm uses the time when a page is to be used. If we use the recent past as an approximation of the near future, then we can replace the page that has not been used for the longest period of time (Figure 21). This approach is the least-recently-used (LRU) algorithm.

LRU replacement associates with each page the time of that page's last use. When a page must be replaced, LRU chooses the page that has not been used for the longest period of time. We can think of this strategy as the optimal page-replacement algorithm looking backward in time, rather than forward. (Strangely, if we let SR be the reverse of a reference string S, then the page-fault rate for the OPT algorithm on S is the same as the page-fault rate for the OPT algorithm on SR. Similarly, the page-fault rate for the LRU algorithm on S is the same as the page-fault rate for the LRU algorithm on SR.)

The result of applying LRU replacement to our example reference string is shown in Figure 21. The LRU algorithm produces 12 faults. Notice that the first 5 faults are the same as those for optimal replacement. When the reference to page 4 occurs, however, LRU replacement sees that, of the three frames in memory, page 2 was used least recently. Thus, the LRU algorithm replaces page 2, not knowing that page 2 is about to be used. When it then faults for page 2, the LRU algorithm replaces page 3, since it is now the least recently used of the three pages in memory. Despite these problems, LRU replacement with 12 faults is much better than FIFO replacement with 15.

The LRU policy is often used as a page-replacement algorithm and is considered to be good. The major problem is how to implement LRU replacement. An LRU page-replacement algorithm may require substantial hardware assistance. The problem is to determine an order for the frames defined by the time of last use. Two implementations are feasible:

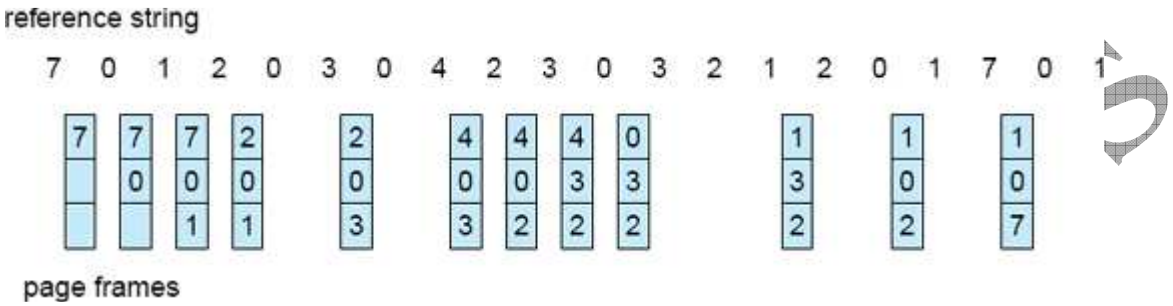


Figure 21: LRU page-replacement algorithm.

- **Counters**: In the simplest case, we associate with each page-table entry a time-of-use field and add to the CPU a logical clock or counter. The clock is incremented for every memory reference. Whenever a reference to a page is made, the contents of the clock register are copied to the time-of-use field in the page-table entry for that page. In this way, we always have the "time" of the last reference to each page. We replace the page with the smallest time value. This scheme requires a search of the page table to find the LRU page and a write to memory (to the time-of-use field in the page table) for each memory access. The times must also be maintained when page tables are changed (due to CPU scheduling). Overflow of the clock must be considered.

- **Stack**: Another approach to implementing LRU replacement is to keep a stack of page numbers. Whenever a page is referenced, it is removed from the stack and put on the top. In this way, the most recently used page is always at the top of the stack and the least recently used page is always at the bottom (Figure 22). Because entries must be removed from the middle of the stack, it is best to implement this approach by using a doubly linked list with a head and tail pointer. Removing a page and putting it on the top of the stack then requires changing six pointers at worst. Each update is a little more expensive, but there is no search for a replacement; the tail pointer points to the bottom of the stack, which is the LRU page. This approach is particularly appropriate for software or microcode implementations of LRU replacement.

Like optimal replacement, LRU replacement does not suffer from Belady's anomaly. Both belong to a class of page-replacement algorithms, called stack algorithms that can never exhibit Belady's anomaly. A stack algorithm is an algorithm for which it can be shown that the set of pages in memory for n frames is always a subset of the set of pages that would be in memory with n + 1 frames. For LRU replacement, the set of pages in memory would be the n most recently referenced pages. If the number of frames is increased, these n pages will still be the most recently referenced and so will still be in memory.

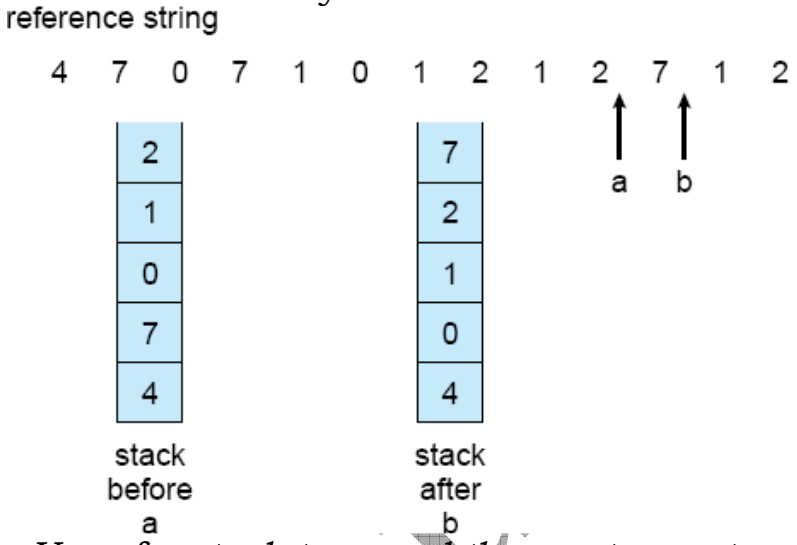


Figure 22: Use of a stack to record the most recent page references.

Note that neither implementation of LRU would be conceivable without hardware assistance beyond the standard TLB registers. The updating of the clock fields or stack must be done for every memory reference. If we were to use an interrupt for every reference to allow software to update such data structures, it would slow every memory reference by a factor of at least ten, hence slowing every user process by a factor of ten. Few systems could tolerate that level of overhead for memory management.

WORKING SET PRINCIPLE

As mentioned, the working-set model is based on the assumption of locality. This model uses a parameter, A, to define the working-set window. The idea is to examine the most recent Δ page references. The set of pages in the most recent Δ page references is the working set (Figure 23). If a page is in active use, it will be in the working set. If it is no longer being used, it will drop from the working set Δ time units after its last reference. Thus, the working set is an approximation of the program's locality.

For example, given the sequence of memory references shown in Figure 23, if Δ = 10 memory references, then the working set at time t1 is {1, 2, 5, 6, 7}. By time t2, the working set has changed to {3,4}.

The accuracy of the working set depends on the selection of Δ. If Δ is too small, it will not encompass the entire locality; if Δ is too large, it may overlap several localities. In the extreme, if Δ is infinite, the working set is the set of pages touched during the process execution.
The most important property of the working set, then, is its size. If we compute the working-set size, WSSi, for each process in the system, we can then consider that

$$D = \Sigma \ WSSi,$$

where D is the total demand for frames. Each process is actively using the pages in its working set. Thus, process i needs WSSi frames. If the total demand is greater than the total number of available frames (D > m), thrashing will occur, because some processes will not have enough frames.

Once Δ has been selected, use of the working-set model is simple. The operating system monitors the working set of each process and allocates to that working set enough frames to provide it with its working-set size. If there are enough extra frames, another process can be initiated. If the sum of the working-set sizes increases, exceeding the total number of available frames, the operating system selects a process to suspend. The process's pages are written out (swapped), and its frames are reallocated to other processes. The suspended process can be restarted later.

This working-set strategy prevents thrashing while keeping the degree of multiprogramming as high as possible. Thus, it optimizes CPU utilization. The difficulty with the working-set model is keeping track of the working set. The working-set window is a moving window. At each memory reference, a new reference appears at one end and the oldest reference drops off the other end. A page is in the working set if it is referenced anywhere in the working-set window.

We can approximate the working-set model with a fixed-interval timer interrupt and a reference bit. For example, assume that Δ equals 10,000 references and that we can cause a timer interrupt every 5,000 references. When we get a timer interrupt, we copy and clear the reference-bit values for each page. Thus, if a page fault occurs, we can

examine the current reference bit and two in-memory bits to determine whether a page was used within the last 10,000 to 15,000 references. If it was used, at least one of these bits will be on. If it has not been used, these bits will be off. Those pages with at least one bit on will be considered to be in the working set. Note that this arrangement is not entirely accurate, because we cannot tell where, within an interval of 5,000, a reference occurred. We can reduce the uncertainty by increasing the number of history bits and the frequency of interrupts (for example, 10 bits and interrupts every 1,000 references). However, the cost to service these more frequent interrupts will be correspondingly higher.

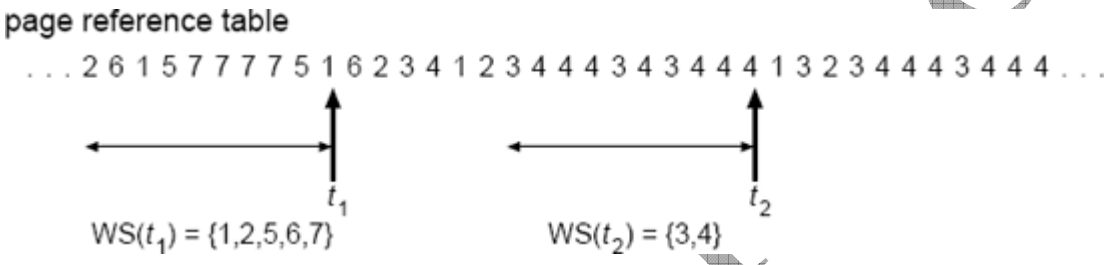


Figure 23: Working-set model.

THRASHING

If the number of frames allocated to a low-priority process falls below the minimum number required by the computer architecture, we must suspend that process's execution. We should then page out its remaining pages, freeing all its allocated frames. This provision introduces a swap-in, swap-out level of intermediate CPU scheduling.

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

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

Cause of Thrashing

Thrashing results in severe performance problems. Consider the following scenario, which is based on the actual behaviour of early paging systems. The operating system monitors CPU utilization. If CPU utilization is too low, we increase the degree of multiprogramming by introducing a new process to the system.

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

The CPU scheduler sees the decreasing CPU utilization and increases the degree of multiprogramming as a result. The new process tries to get started by taking frames from running processes, causing more page faults and a longer queue for the paging device. As a result, CPU utilization drops even further, and the CPU scheduler tries to increase the degree of multiprogramming even more. Thrashing has occurred, and system throughput plunges. The page-fault rate increases tremendously. As a result, the effective memory-access time increases. No work is getting done, because the processes are spending all their time paging.

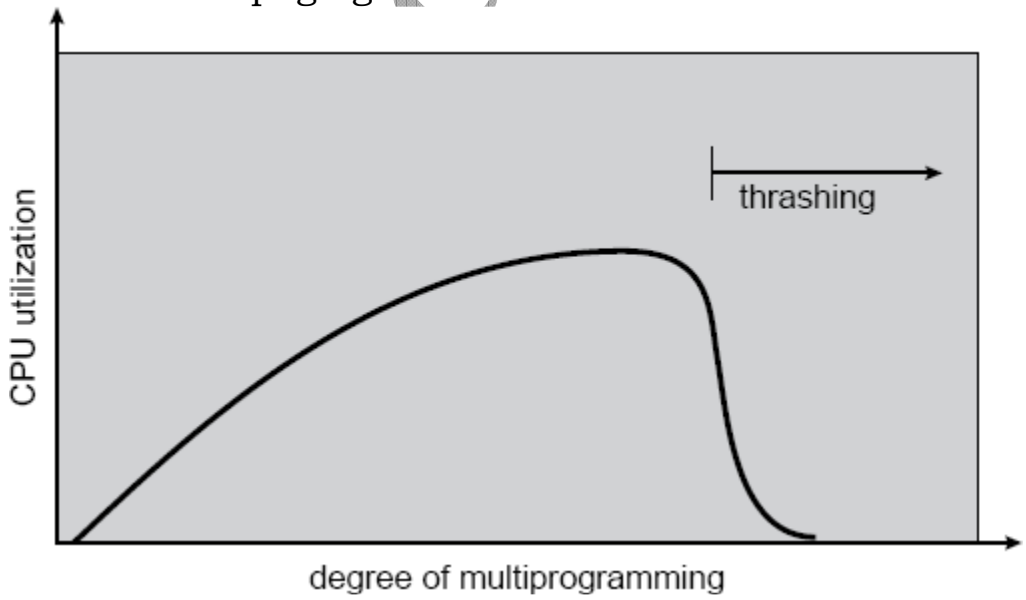


Figure 24: Thrashing.

This phenomenon is illustrated in Figure 24, in which CPU utilization is plotted against the degree of multiprogramming. As the degree of multiprogramming increases, CPU utilization also increases, although more slowly, until a maximum is reached. If the degree of multiprogramming is increased even further, thrashing sets in, and CPU utilization drops sharply. At this point, to increase CPU utilization and stop thrashing, we must decrease the degree of multiprogramming.

We can limit the effects of thrashing by using a local replacement algorithm (or priority replacement algorithm). With local replacement, if one process starts thrashing, it cannot steal frames from another process and cause the latter to thrash as well. However, the problem is not entirely solved. If processes are thrashing, they will be in the queue for the paging device most of the time. The average service time for a page fault will increase because of the longer average queue for the paging device. Thus, the effective access time will increase even for a process that is not thrashing.

To prevent thrashing, we must provide a process with as many frames as it needs. But how do we know how many frames it "needs"? There are several techniques. The working-set strategy starts by looking at how many frames a process is actually using. This approach defines the locality model of process execution. The locality model states that, as a process executes, it moves from locality to locality. A locality is a set of pages that are actively used together. A program is generally composed of several different localities, which may overlap.