SYLLABUS

Objective: Enable the student to get sufficient knowledge on various system resources.

Unit – I Operating System Basics
Basic Concepts of Operating System - Services of Operating System-Classification of Operating System- Architecture and Design of an Operating System-Process Management - Introduction to Process-Process State -PCB - Process Scheduling - Interprocess Communication

Unit –II Operating System Scheduling
CPU Scheduling**:** Introduction - Types of CPU Scheduler - Scheduling Criteria - Scheduling Algorithms - FCFS Scheduling – SJF Scheduling;-Priority Scheduling - Round-Robin Scheduling- Multilevel Queue Scheduling - Deadlock - Basic Concept of Deadlock-Deadlock Prevention - Deadlock Avoidance- Deadlock - Detection and Recovery

Unit- III Memory management
Memory Management - Basic Concept of Memory - Address Binding; Logical and Physical Address Space- Memory Partitioning - Memory Allocation-Protection-Fragmentation and Compaction

Unit – IV Swapping
Swapping- Using Bitmaps - Using Linked Lists- Paging-Mapping of Pages to Frames - Hierarchical Page Tables- Segmentation - Virtual Memory - Basic Concept of Virtual Memory- Demand Paging - Transaction Look aside Buffer (TLB) - Inverted Page Table-Page Replacement Algorithms

Unit –V File Management
File Management - Basic Concept of File-Directory Structure-File Protection-Allocation Methods – Various Disk Scheduling algorithms

Text Books:

1. Abraham Silberschatz Peter B. Galvin, G. Gagne, "Operating System Concepts", Sixth Edition, Addison Wesley Publishing Co., 2003.

Reference Books:

1. Operating systems - Internals and Design Principles, W. Stallings, 6th Edition, Pearson
2. Operating System - Willam-Stallings Fourth Edition, Pearson Education, 2003.
3. Modern Operating Systems – Andrew S. Tanenbaum, 3 rd Edition, PHI 3.
4. Operating Systems: A Spiral Approach – Elmasri, Carrick, Levine, TMH Edition
5. Operating Systems – Flynn, McHoes, Cengage Learning
6. Operating Systems – Pabitra Pal Choudhury, PHI
7. Operating Systems – H.M. Deitel, P. J. Deitel, D. R. Choffnes, 3 rd Edition, Pearson
8. Operating Systems: Three Easy Pieces - Andrea C. Arpaci-Dusseau and Remzi H. Arpaci-Dusseau
9. Understanding operating systems - Ida Flynn
10. Operating Systems: A Modern Perspective - Gary J Nutt
11. Operating Systems: Design and Implementation - Albert S. Woodhull and Andrew S. Tanenbaum

UNIT 1. OPERATING SYSTEM BASICS

INTRODUCTION

DEFINITION

"An **operating system** is a program that manages a computer's hardware. It also provides a basis for application programs and acts as an intermediary between the computer user and the computer hardware".

The purpose of an operating system is to provide an environment in which a user can execute programs in a convenient and efficient manner. An operating system is large and complex, it must be created piece by piece. Each of these pieces should be a well-delineated portion of the system, with carefully defined inputs, outputs, and functions. An operating system is software that manages the computer hardware. The hardware must provide appropriate mechanisms to ensure the correct operation of the computer system and to prevent user programs from interfering with the proper operation of the system.

WORKING OF OPERATING SYSTEM

A computer system can be divided into four components with hardware, the operating system, the application programs, and the users as shown in figure 1.1.

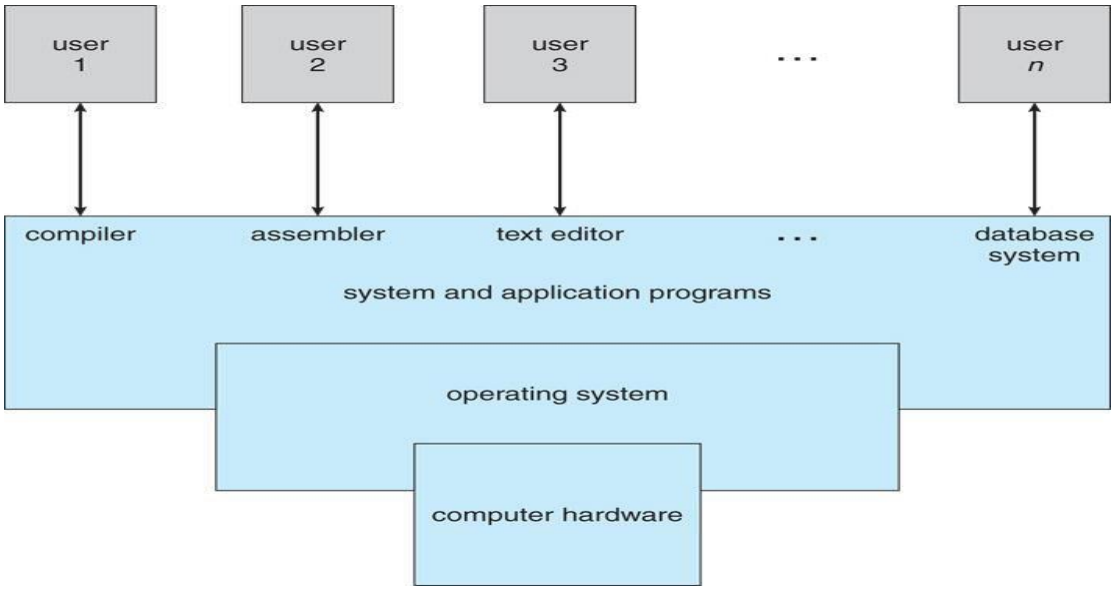


Figure 1.1 - Abstract View of the Components of a Computer System

The hardware **Central Processing Unit (CPU)**, the **Memory**, and the **Input / Output (I/O) devices** provides the basic computing resources for the system. The **Application**

Programs such as word processors, spreadsheets, compilers, and Web browsers define the ways in which these resources are used to solve users' computing problems. A computer system as consisting of hardware, software, and data. The operating system provides the means for proper use of these resources in the operation of the computer system.

COMPUTER SYSTEM ORGANIZATION

A modern general-purpose computer system consists of one or more CPUs and a number of device controllers connected through a common bus that provides access to shared memory Figure 1.2. Each device controller is in charge of a specific type of device (for example, disk drives, audio devices, or video displays). The CPU and the device controllers can execute in parallel, competing for memory cycles. To ensure orderly access to the shared memory, a memory controller synchronizes access to the memory.

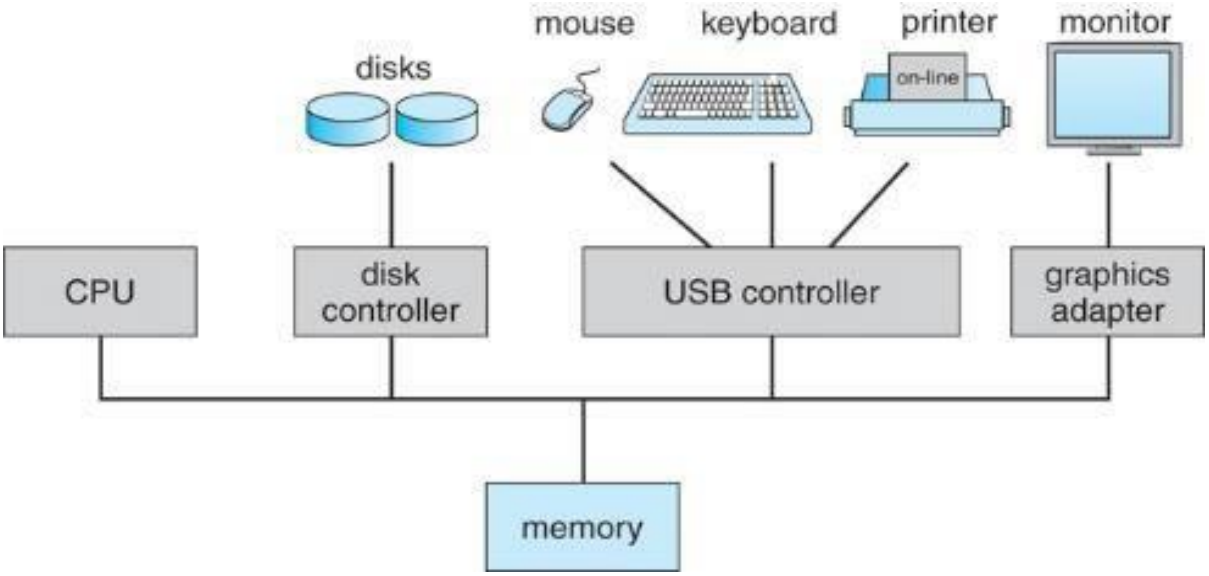


Figure 1.2 - A Modern Computer System

When the system is powered up or rebooted it needs to have an initial program to run. This initial program, or **bootstrap program**, tends to be simple. Typically, it is stored within the computer hardware in read-only memory (**ROM**) or electrically erasable programmable read-only memory (**EEPROM**), known by the general term **firmware**. It initializes all aspects of the system, from CPU registers to device controllers to memory contents. The bootstrap program must know how to load the operating system and how to start executing that system.

The bootstrap program must locate the operating-system kernel and load it into memory. Once the kernel is loaded and executing, it can start providing services to the system and its

users. Some services are provided outside of the kernel, by system programs that are loaded into memory at boot time to become **system processes**, or **system daemons** that run the entire time the kernel is running.

A **Kernel** is a central component of an operating system. It acts as an interface between the user applications and the hardware.

Booting is a start-up sequence that starts the operating system of a computer when it is turned on.

A **Bootstrap** is the program that initializes the Operating System (**OS**) during start-up which initiates the smaller program that executed a larger program such as the **OS**.

The occurrence of an event is usually signalled by an **interrupt** from either the hardware or the software. Hardware may trigger an interrupt at any time by sending a signal to the CPU, usually by way of the system bus. Software may trigger an interrupt by executing a special operation called a **system call** (also called a **monitor call**). When the CPU is interrupted, it stops what it is doing and immediately transfers execution to a fixed location. The fixed location usually contains the starting address where the service routine for the interrupt is located. The interrupt service routine executes; on completion, the CPU resumes the interrupted computation.

Interrupt is a signal that gets the attention of the CPU and is usually generated when I/O is required".

A **System Call** is the programmatic way in which a computer program requests a service from the kernel of the operating system it is executed on. It provides an interface between a process and operating system to allow user-level processes to

STORAGE STRUCTURES, DEFINITIONS AND NOTATIONS

All forms of memory provide an array of bytes. Each byte has its own address. Interaction is achieved through a sequence of load or store instructions to specific memory addresses. The load instruction moves a byte or word from main memory to an internal register within the CPU, whereas the store instruction moves the content of a register to main memory.

Bit – basic unit 1 or 0

Byte – 8 bits

Kilobyte (KB) – 1024 bytes

Megabyte (MB) -- 1024^2 bytes

Gigabyte (GB) – 1024^3 bytes

Terabyte (TB) – 1024^4 bytes

Petabyte (PB) – 1024^5 bytes

A typical instruction–execution cycle, as executed on a system with a Von Neumann architecture, first fetches an instruction from memory and stores that instruction in the instruction register. The instruction is then decoded and may cause operands to be fetched from memory and stored in some internal register. After the instruction on the operands has been executed, the result may be stored back in memory. Each storage system provides the basic functions of storing a datum and holding that datum until it is retrieved at a later time. The main differences among the various storage systems lie in speed, cost, size, and volatility. The wide variety of storage systems can be organized in a hierarchy Figure1.3 according to speed and cost.

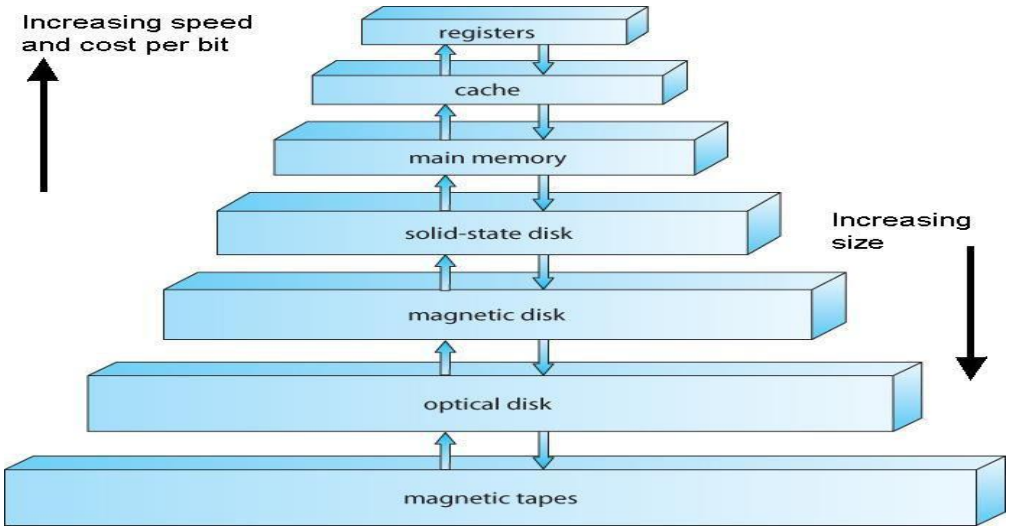


Figure 1.3 - Storage-device hierarchy

I/O STRUCTURES

A general-purpose computer system consists of CPUs and multiple device controllers that are connected through a common bus. Each device controller is in charge of a specific type of device. Depending on the controller, more than one device may be attached. Seven or more devices can be attached to the **small computer-systems interface (SCSI)** controller. A

device controller maintains some local buffer storage and a set of special-purpose registers. The device controller is responsible for moving the data between the peripheral devices that it controls and its local buffer storage. Typically, operating systems have a **device driver** for each device controller. This device driver understands the device controller and provides the rest of the operating system with a uniform interface to the device. To start an I/O operation, the device driver loads the appropriate registers within the device controller.

The device controller, in turn, examines the contents of these registers to determine what action to take (such as "read a character from the keyboard"). The controller starts the transfer of data from the device to its local buffer. Once the transfer of data is complete, the device controller informs the device driver via an interrupt that it has finished its operation. The device driver then returns control to the operating system, possibly returning the data or a pointer to the data if the operation was a read. For other operations, the device driver returns status information. This form of interrupt-driven I/O is fine for moving small amounts of data but can produce high overhead when used for bulk data movement such as disk I/O.

Direct Memory Access (**DMA**) is a method that allows an input/output (I/O) device to send or receive data directly to or from the main memory, bypassing the CPU to speed up memory **operations**. The process is managed by a chip known as a **DMA** controller **(DMAC)**.

To solve this problem, **Direct Memory Access (DMA)** is used. After setting up buffers, pointers, and counters for the I/O device, the device controller transfers an entire block of data directly to or from its own buffer storage to memory, with no intervention by the CPU. Only one interrupt is generated per block, to tell the device driver that the operation has completed, rather than the one interrupt per byte generated for low-speed devices. While the device controller is performing these operations, the CPU is available to accomplish other work. Some high-end systems use switch rather than bus architecture. On these systems, multiple components can talk to other components concurrently, rather than competing for cycles on a shared bus. The DMA is shown more effectively in Figure 1.4 which shows the interplay of all components of a computer system.

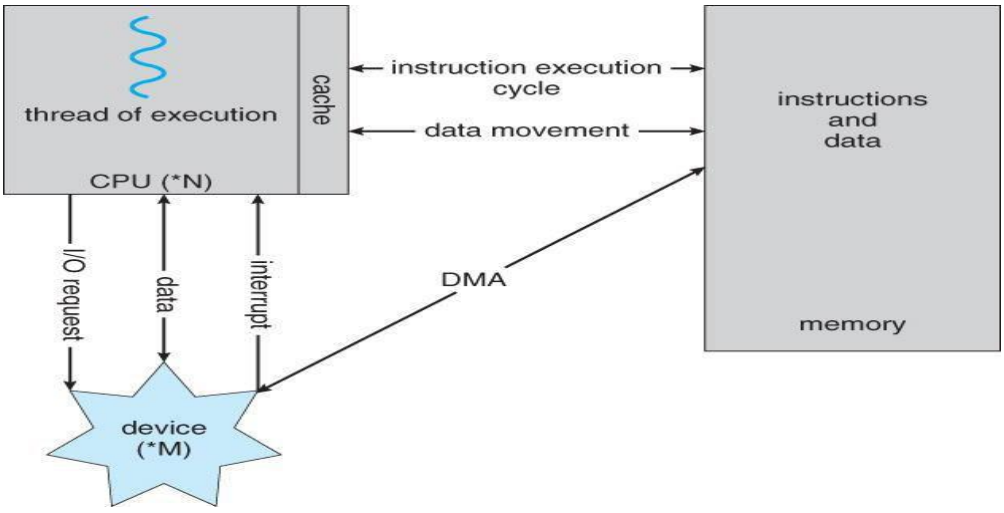


Figure 1.4 Working of computer systems

Operating systems can be classified as follows:

Multi-user: is the one that concede two or more users to use their programs at the same time. Some of O.S permits hundreds or even thousands of users simultaneously.

Single-User: just allows one user to use the programs at one time.

Multiprocessor: Supports opening the same program more than just in one CPU.

Multitasking: Allows multiple programs running at the same time.

Single-tasking: Allows different parts of a single program running at any one time.

Real time: Responds to input instantly. Operating systems such as DOS and UNIX, do not work in real time.

Here is a list of common services offered by an almost all operating systems:

- User Interface
- Program Execution
- File system manipulation
- Input / Output Operations
- Communication
- Resource Allocation
- Error Detection
- Accounting
- Security and protection

COMPUTER-SYSTEM ARCHITECTURE

A computer system can be organized in a number of different ways, which we can categorize roughly according to the number of general-purpose processors used. Different types of Operating Systems for Different Kinds of Computer Environments are classified as

- Single processor system
- Multiprocessor system
- Clustered systems

SINGLE PROCESSOR SYSTEM

On a single processor system, there is one main CPU capable of executing a general-purpose instruction set, including instructions from user processes. Almost all single processor systems have other special-purpose processors as well. They may come in the form of device-specific processors, such as disk, keyboard, and graphics controllers; or, on mainframes, they may come in the form of more general-purpose processors, such as I/O processors that move data rapidly among the components of the system. All of these special purpose processors run a limited instruction set and do not run user processes.

MULTIPROCESSOR SYSTEM

Multiprocessor Systems (also known as **parallel systems** or **multicore systems** have two or more processors for communication, sharing the computer bus and the clock, memory, and peripheral devices. Multiprocessor systems first appeared in servers and now it have migrated to desktop and laptop systems. Multiple processors have appeared on mobile devices such as smartphones and tablet computers also. Multiprocessor systems have three main advantages:

1. Increased throughput. By increasing the number of processors, we expect to get more work done in less time.

2. Economy of scale. Multiprocessor systems can cost less than equivalent multiple single-processor systems, because they can share peripherals, mass storage, and power supplies. If several programs operate on the same set of data, it is cheaper to store those data on one disk and to have all the processors share them than to have many computers with local disks and many copies of the data.

3. Increased reliability. If functions can be distributed properly among several processors, then the failure of one processor will not halt the system, only slow it down. If we have ten processors and one fails, then each of the remaining nine processors can pick up a share of

the work of the failed processor. Thus, the entire system runs only 10 percent slower, rather than failing altogether. Increased reliability of a computer system is crucial in many applications. The ability to continue providing service proportional to the level of surviving hardware is called **graceful degradation**. Some systems go beyond graceful degradation and are called **fault tolerant**, because they can suffer a failure of any single component and still continue operation. Fault tolerance requires a mechanism to allow the failure to be detected, diagnosed, and, if possible, corrected.

The multiprocessor systems are classified into two categories and they are

- Asymmetric multiprocessor
- Symmetric multiprocessor

Asymmetric multiprocessor is a processor in which each processor is assigned a specific task. This scheme defines a boss–worker relationship. The boss processor schedules and allocates work to the worker processors.

Symmetric multiprocessor (SMP), in which each processor performs all tasks within the operating system. SMP means that all processors are peers; no boss–worker relationship exists between processors. Figure 1.5 illustrates a typical SMP architecture. Multiprocessing adds CPUs to increase computing power. If the CPU has an integrated memory controller, then adding CPUs can also increase the amount of memory addressable in the system.

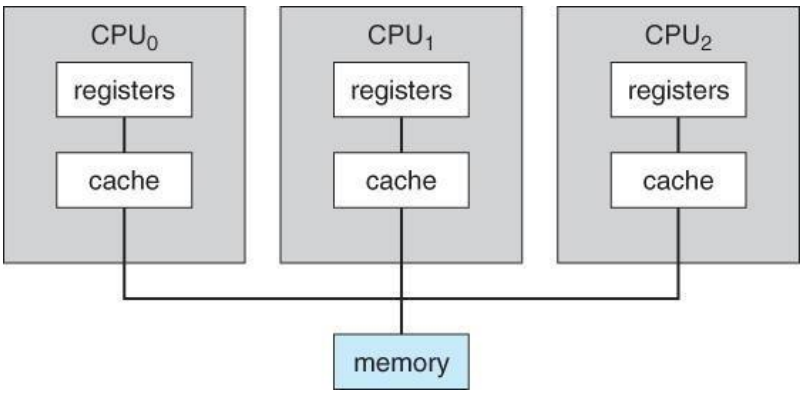


Figure 1.5 - Symmetric multiprocessing architecture

CLUSTERED SYSTEMS

Another type of multiprocessor system is a **clustered system** in Figure 1.6, which gathers together multiple CPUs. Clustered systems differ from the multiprocessor systems which are composed of two or more individual systems—or nodes—joined together. Such systems are considered **loosely coupled**. Each node may be a single processor system or a multicore system.

Clustering is usually used to provide **high-availability** service—that is, service will continue even if one or more systems in the cluster fail. Generally, we obtain high availability by adding a level of redundancy in the system. A layer of cluster software runs on the cluster nodes. Each node can monitor one or more of the others (over the LAN). If the monitored machine fails the monitoring machine can take ownership of its storage and restart the applications that were running on the failed machine. The users and clients of the applications see only a brief interruption of service. Clustering can be structured asymmetrically or symmetrically. In **asymmetric clustering**, one machine is in **hot-standby mode** while the other is running the applications. In **symmetric clustering**, two or more hosts are running applications and are monitoring each other.

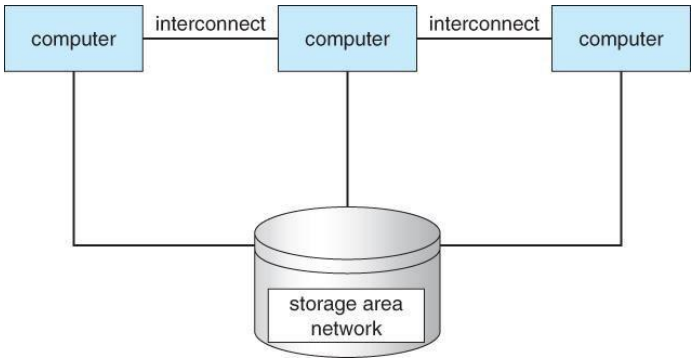


Figure 1.6 - General structure of a clustered system

OPERATING-SYSTEM STRUCTURE

One of the most important aspects of operating systems is the ability to multi program. A single program cannot be kept either in the CPU or in the I/O devices as the processor will be busy at all times. Single users frequently have multiple programs running. **Multiprogramming** increases CPU utilization by organizing jobs (code and data) so that the CPU always has one to execute.

The operating system keeps several jobs in memory simultaneously (Figure 1.7). Since main memory is too small to accommodate all jobs, the jobs are kept initially on the disk in the **job pool**. This pool consists of all processes residing on disk awaiting allocation of main memory. The set of jobs in memory can be a subset of the jobs kept in the job pool. The operating system picks and begins to execute one of the jobs in memory. Eventually, the job may have to wait for some task, such as an I/O operation, to complete.

In a non-multiprogrammed system, the CPU would sit idle. In a multiprogrammed system, the operating system simply switches to, and executes, another job. When that job

needs to wait, the CPU switches to another job, and so on. Eventually, the first job finishes waiting and gets the CPU back.

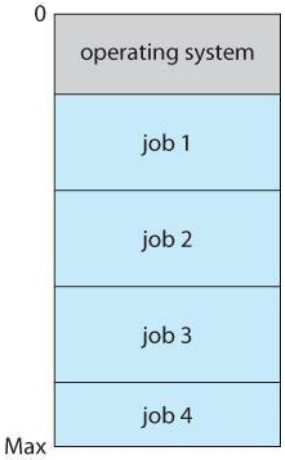


Figure 1.7 - Memory layout for a multiprogramming system

A time-sharing (multi-user multi-tasking) OS requires:

- Memory management
- Process management
- Job scheduling
- Resource allocation strategies
- Swap space / virtual memory in physical memory
- Interrupt handling
- File system management
- Protection and security
- Inter-process communications

Time sharing (or **multitasking**) is a logical extension of multiprogramming. In time-sharing systems, the CPU executes multiple jobs by switching among them, but the switches occur so frequently that the users can interact with each program while it is running. Time sharing requires an **interactive** computer system, which provides direct communication between the user and the system. The user gives instructions to the operating system or to a program directly, using a input device such as a keyboard, mouse, touch pad, or touch screen, and waits for immediate results on an output device. Accordingly, the **response time** should be short—typically less than one second. A time-shared operating system allows many users to share the computer simultaneously. A time-shared operating 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.

A program loaded into memory and executing is called a **process**.

Time sharing and multiprogramming require that several jobs be kept simultaneously in 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 choose among them. Making this decision involves **Job Scheduling**. When the operating system selects a job from the job pool, it loads that job into memory for execution. Having several programs in memory at the same time requires some form of memory management. In addition, if several jobs are ready to run at the same time, the system must choose which job will run first. Making this decision is **CPU Scheduling**. Finally, running multiple jobs concurrently requires that their ability to affect one another be limited in all phases of the operating system, including process scheduling, disk storage, and memory management.

The main advantage of the virtual-memory scheme is that it enables users to run programs that are larger than actual **Physical Memory**. Further, it abstracts main memory into a large, uniform array of storage, separating **Logical Memory** as viewed by the user from physical memory. This arrangement frees programmers from concern over memory-storage limitations.

OPERATING-SYSTEM OPERATIONS

Modern operating systems are **interrupt driven**. If there are no processes to execute, no I/O devices to service, and no users to whom to respond, an operating system will sit quietly, waiting for something to happen. Events are almost always signalled by the occurrence of an interrupt or a trap. A **trap** (or an **exception**) is a software-generated interrupt caused either by an error (for example, division by zero or invalid memory access) or by a specific request from a user program that an operating-system service be performed.

The interrupt-driven nature of an operating system defines that system's general structure. For each type of interrupt, separate segments of code in the operating system determine what action should be taken. An interrupt service routine is provided to deal with the interrupt. Since the operating system and the users share the hardware and software resources of the computer system, we need to make sure that an error in a user program could cause problems only for the one program running.

With sharing, many processes could be adversely affected by a bug in one program. For example, if a process gets stuck in an infinite loop, this loop could prevent the correct operation of many other processes. More subtle errors can occur in a multiprogramming system, where one erroneous program might modify another program, the data of another program, or even the operating system itself. Without protection against these sorts of errors, either the computer must execute only one process at a time or all output must be suspect. A properly designed operating system must ensure that an incorrect (or malicious) program cannot cause other programs to execute incorrectly.

DUAL-MODE AND MULTIMODE OPERATION

In order to ensure the proper execution of the operating system, we must be able to distinguish between the execution of operating-system code and user defined code. The approach taken by most computer systems is to provide hardware support that allows us to differentiate among various modes of execution.

- User mode when executing harmless code in user applications
- Kernel mode (a.k.a. system mode, supervisor mode, privileged mode) when executing potentially dangerous code in the system kernel.
- Certain machine instructions (privileged instructions) can only be executed in kernel mode.
- Kernel mode can only be entered by making system calls. User code cannot flip the mode switch.
- Modern computers support dual-mode operation in hardware, and therefore most modern OS support dual-mode operation.
- The concept of modes can be extended beyond two, requiring more than a single mode bit CPUs that support virtualization use one of these extra bits to indicate when the virtual machine manager, VMM, is in control of the system. The VMM has more privileges than ordinary user programs, but not so many as the full kernel.
- System calls are typically implemented in the form of software interrupts, which causes the hardware's interrupt handler to transfer control over to an appropriate interrupt handler, which is part of the operating system, switching the mode bit to kernel mode in the process.

- The interrupt handler checks exactly which interrupt was generated, checks additional parameters (generally passed through registers) if appropriate, and then calls the appropriate kernel service routine to handle the service requested by the system call.
- User programs' attempts to execute illegal instructions (privileged or non-existent instructions), or to access forbidden memory areas, also generate software interrupts, which are trapped by the interrupt handler and control is transferred to the OS, which issues an appropriate error message, possibly dumps data to a log file for later analysis, and then terminates the offending program.

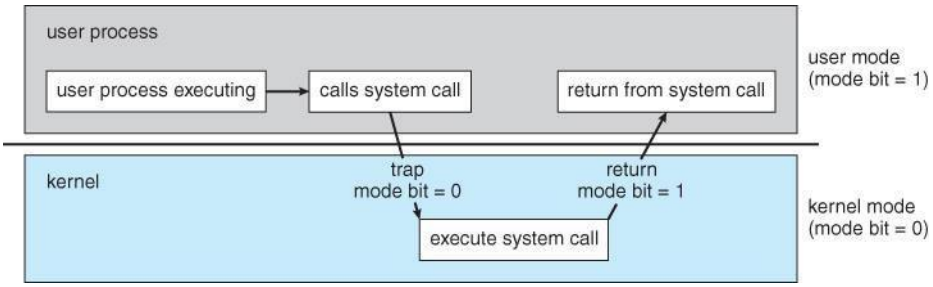


Figure 1.8 - Transition from user to kernel mode

We need two separate ***modes*** of operation: **user mode** and **kernel mode** (also called **supervisor mode**, **system mode**, or **privileged mode**). A bit, called the **mode bit**, is added to the hardware of the computer to indicate the current mode: kernel (0) or user (1). With the mode bit, we can distinguish between a task that is executed on behalf of the operating system and one that is executed on behalf of the user. When the computer system is executing on behalf of a user application, the system is in user mode.

However, when a user application requests a service from the operating system (via a system call), the system must transition from user to kernel mode to fulfil the request. This is shown in Figure 1.8. This architectural enhancement is useful for many other aspects of system operation as well. At system boot time, the hardware starts in kernel mode. The operating system is then loaded and starts user applications in user mode. Whenever a trap or interrupt occurs, the hardware switches from user mode to kernel mode (that is, changes the state of the mode bit to 0). Thus, whenever the operating system gains control of the computer, it is in kernel mode. The system always switches to user mode (by setting the mode bit to 1) before passing control to a user program. The dual mode of operation provides us with the means for protecting the operating system from errant users—and errant users from one another. We accomplish this protection by designating some of the machine instructions that may cause harm as **privileged instructions**.

The hardware allows privileged instructions to be executed only in kernel mode. If an attempt is made to execute a privileged instruction in user mode, the hardware does not execute the instruction but rather treats it as illegal and traps it to the operating system.

TIMER

The operating system maintains control over the CPU. We cannot allow a user program to get stuck in an infinite loop or to fail to call system services and never return control to the operating system. To accomplish this goal, we can use a **timer**. A timer can be set to interrupt the computer after a specified period. The period may be fixed (for example, 1/60 second) or variable (for example, from 1 millisecond to 1 second).

A **variable timer** is generally implemented by a fixed-rate clock and a counter. The operating system sets the counter. Every time the clock ticks, the counter is decremented. When the counter reaches 0, an interrupt occurs. For instance, a 10-bit counter with a 1-millisecond clock allows interrupts at intervals from 1 millisecond to 1,024 milliseconds, in steps of 1 millisecond. Before turning over control to the user, the operating system ensures that the timer is set to interrupt. If the timer interrupts, control transfers automatically to the operating system, which may treat the interrupt as a fatal error or may give the program more time. Clearly, instructions that modify the content of the timer are privileged. We can use the timer to prevent a user program from running too long.

A simple technique is to initialize a counter with the amount of time that a program is allowed to run. A program with a 7-minute time limit, for example, would have its counter initialized to 420. Every second, the timer interrupts, and the counter is decremented by 1. As long as the counter is positive, control is returned to the user program. When the counter becomes negative, the operating system terminates the program for exceeding the assigned time limit.

PROCESS MANAGEMENT

A program does nothing unless its instructions are executed by a CPU. A program in execution, as mentioned, is a process. A time-shared user program such as a compiler is a process. A process needs certain resources—including CPU time, memory, files, and I/O devices—to accomplish its task. These resources are either given to the process when it is created or allocated to it while it is running.

A program by itself is not a process. A program is a ***passive*** entity, like the contents of a file stored on disk, whereas a process is an ***active*** entity.

A single-threaded process has one **program counter** specifying the next instruction to execute. The execution of such a process must be sequential. The CPU executes one instruction of the process after another, until the process completes.

A multithreaded process has multiple program counters, each pointing to the next instruction to execute for a given thread. A process is the unit of work in a system. A system consists of a collection of processes, some of which are operating-system processes (those that execute system code) and the rest of which are user processes (those that execute user code). All these processes can potentially execute concurrently—by multiplexing on a single CPU.

An OS is responsible for the following tasks with regards to process management:

- Creating and deleting both user and system processes
- Ensuring that each process receives its necessary resources, without interfering with other processes.
- Suspending and resuming processes
- Process synchronization and communication
- Deadlock handling

MEMORY MANAGEMENT

The main memory is central to the operation of a modern computer system. Main memory is a large array of bytes, ranging in size from hundreds of thousands to billions. Each byte has its own address. Main memory is a repository of quickly accessible data shared by the CPU and I/O devices. The central processor reads instructions from main memory during the instruction-fetch cycle and both reads and writes data from main memory during the data-fetch cycle.

For a program to be executed, it must be mapped to absolute addresses and loaded into memory. As the program executes, it accesses program instructions and data from memory by generating these absolute addresses. Eventually, the program terminates, its memory space is declared available, and the next program can be loaded and executed. To improve both the utilization of the CPU and the speed of the computer's response to its users, general-purpose computers must keep several programs in memory, creating a need for memory management.

An OS is responsible for the following tasks with regards to memory management:

- Keeping track of which blocks of memory are currently in use, and by which processes.

- Determining which blocks of code and data to move into and out of memory, and when.
- Allocating and deallocating memory as needed. (E.g. new, malloc)

STORAGE MANAGEMENT

FILE-SYSTEM MANAGEMENT

An OS is responsible for the following tasks with regards to filesystem management:

- Creating and deleting files and directories
- Supporting primitives for manipulating files and directories. (open, flush, etc.)
- Mapping files onto secondary storage.
- Backing up files onto stable permanent storage media.

MASS-STORAGE MANAGEMENT

An OS is responsible for the following tasks with regards to mass-storage management:

- Free disk space management
- Storage allocation
- Disk scheduling

Note the trade-offs regarding size, speed, longevity, security, and re-writable between different mass storage devices, including floppy disks, hard disks, tape drives, CDs, DVDs, etc.

CACHING

- There are many cases in which a smaller higher-speed storage space serves as a cache, or temporary storage, for some of the most frequently needed portions of larger slower storage areas.
- The hierarchy of memory storage ranges from CPU registers to hard drives and external storage is reflected in Table 1.
- The OS is responsible for determining what information to store in what level of cache, and when to transfer data from one level to another.

Table 1. Performance of various levels of storage

Level	1	2	3	4	5
Name	registers	cache	main memory	solid state disk	magnetic disk
Typical size	< 1 KB	< 16MB	< 64GB	< 1 TB	< 10 TB
Implementation technology	custom memory with multiple ports CMOS	on-chip or off-chip CMOS SRAM	CMOS SRAM	flash memory	magnetic disk
Access time (ns)	0.25 - 0.5	0.5 - 25	80 - 250	25,000 - 50,000	5,000,000
Bandwidth (MB/sec)	20,000 - 100,000	5,000 - 10,000	1,000 - 5,000	500	20 - 150
Managed by	compiler	hardware	operating system	operating system	operating system
Backed by	cache	main memory	disk	disk	disk or tape

- The proper choice of cache management can have a profound impact on system performance.
- Data read in from disk follows a migration path from the hard drive to main memory, then to the CPU cache, and finally to the registers before it can be used, while data being written follows the reverse path. Each step (other than the registers) will typically fetch more data than is immediately needed, and cache the excess in order to satisfy future requests faster. For writing, small amounts of data are frequently buffered until there is enough to fill an entire "block" on the next output device in the chain.
- The issues get more complicated when multiple processes (or worse multiple computers) access common data, as it is important to ensure that every access reaches the most up-to-date copy of the cached data (amongst several copies in different cache levels.)

In a hierarchical storage structure, the same data may appear in different levels of the storage system. For example, suppose that an integer A that is to be incremented by 1 is located in file B, and file resides on magnetic disk. The increment operation proceeds by first issuing an I/O operation to copy the disk block on which A resides to main memory.

This operation is followed by copying A to the cache and to an internal register. Thus, the copy of A appears in several places: on the magnetic disk, in main memory, in the cache, and in an internal register shown in Figure 1.9. Once the increment takes place in the internal register, the value of A differs in the various storage systems. The value of A becomes the same only after the new value of A is written from the internal register back to the magnetic disk.

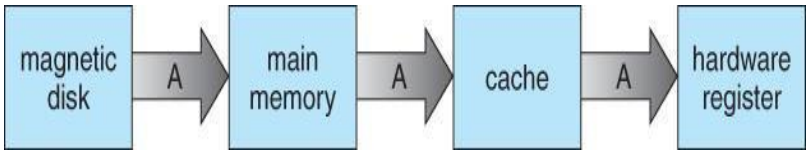


Figure 1.9 - Migration of integer A from disk to register

I/O SYSTEMS

The I/O subsystem consists of several components:

- A memory-management component that includes buffering, caching, and spooling.
- A general device-driver interface.
- Drivers for specific hardware devices.
- (UNIX implements multiple device interfaces for many types of devices, one for accessing the device character by character and one for accessing the device block by block. These can be seen by doing a long listing of /dev, and looking for a "c" or "b" in the first position. You will also note that the "size" field contains two numbers, known as the major and minor device numbers, instead of the normal one. The major number signifies which device driver handles I/O for this device, and the minor number is a parameter passed to the driver to let it know which specific device is being accessed. Where a device can be accessed as either a block or character device, the minor numbers for the two options usually differ by a single bit.)

PROTECTION AND SECURITY

- ***Protection*** involves ensuring that no process access or interfere with resources to which they are not entitled, either by design or by accident. (E.g. "protection faults" when pointer variables are misused.)
- ***Security*** involves protecting the system from deliberate attacks, either from legitimate users of the system attempting to gain unauthorized access and privileges, or external attackers attempting to access or damage the system.

KERNAL DATA STRUCTURES

LISTS, STACKS, AND QUEUES

An array is a simple data structure in which each element can be accessed directly. Each item in an array can be accessed directly, the items in a list must be accessed in a particular order. That is, a **list** represents a collection of data values as a sequence. The most common method for implementing this structure is a **linked list**, in which items are linked to one another. Linked lists are of several types:

• In a ***singly linked list,*** each item points to its successor, as illustrated in Figure 1.10.

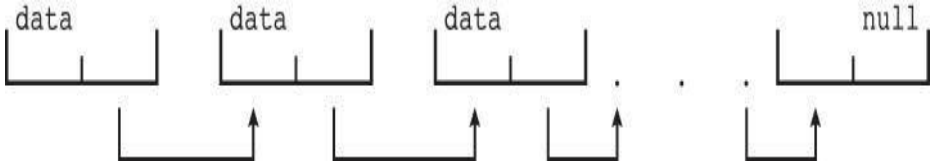


Figure 1.10 - Singly linked list

• In a ***doubly linked list,*** a given item can refer either to its predecessor or to its successor, as illustrated in Figure 1.11.

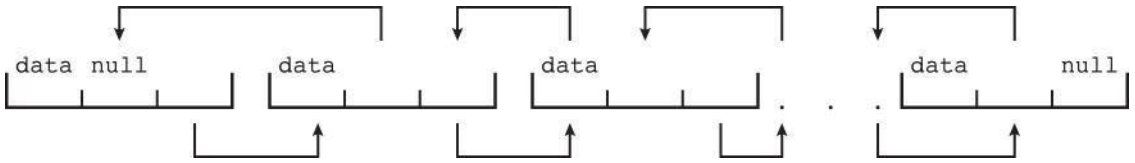


Figure 1.11 - Doubly linked list

• In a ***circularly linked list,*** the last element in the list refers to the first element, rather than to null, as illustrated in Figure 1.12.

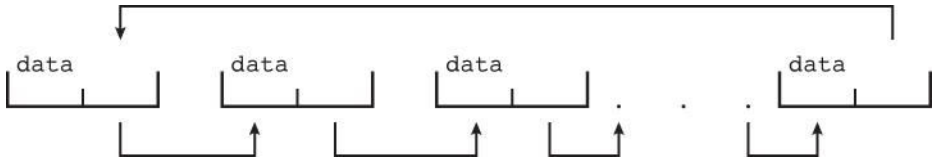


Figure 1.12 - Circularly linked list

Linked lists accommodate items of varying sizes and allow easy insertion and deletion of items.

A **stack** is a sequentially ordered data structure that uses the last in, first out **(LIFO)** principle for adding and removing items, meaning that the last item placed onto a stack is the first item removed. The operations for inserting and removing items from a stack are known as ***push*** and ***pop***, respectively. An operating system often uses a stack when invoking function calls. Parameters, local variables, and the return address are pushed onto the stack when a function is called; returning from the function call pops those items off the stack.

A **queue**, in contrast, is a sequentially ordered data structure that uses the first in, first out **(FIFO)** principle: items are removed from a queue in the order in which they were inserted. There are many everyday examples of queues, including shoppers waiting in a checkout line at a store and cars waiting in line at a traffic signal. Queues are also quite common in operating systems—jobs that are sent to a printer are typically printed in the order in which they were submitted.

TREES

A **tree** is a data structure that can be used to represent data hierarchically. Data values in a tree structure are linked through parent–child relationships. In a **general tree**, a parent may have an unlimited number of children. In a **binary tree**, a parent may have at most two children, which we term the ***left child*** and the ***right child***. A **binary search tree** additionally requires an ordering between the parent's two children in which *le f t child <= right child*. Figure 1.13 provides an example of a binary search tree.

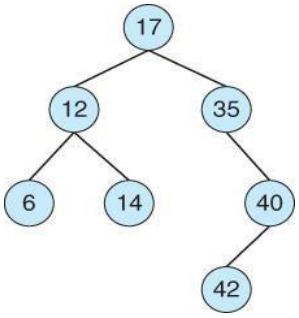


Figure 1.13 - Binary search trees

COMPUTING ENVIRONMENTS

Operating systems are used in a variety of computing environments such as

- Traditional computing
- Distributed computing
- Mobile computing
- Client server computing
- Peer to peer computing
- Virtualization
- Cloud computing
- Real time embedded systems

TRADITIONAL COMPUTING

PCs connected to a network, with servers providing file and print services. Remote access was awkward, and portability was achieved by use of laptop computers. Terminals attached to mainframes were prevalent at many companies as well, with even fewer remote access and portability options. The current trend is toward providing more ways to access these computing environments. Web technologies and increasing WAN bandwidth are stretching the boundaries of traditional computing. Companies establish **portals**, which

provide Web accessibility to their internal servers. **Network computers** (or **thin clients**) which are essentially terminals that understand web-based computing—are used in place of traditional workstations where more security or easier maintenance is desired.

MOBILE COMPUTING

- Computing on small handheld devices such as smart phones or tablets. (As opposed to laptops, which still fall under traditional computing)
- May take advantage of additional built-in sensors, such as GPS, tilt, compass, and inertial movement.
- Typically connect to the Internet using wireless networking (IEEE 802.11) or cellular telephone technology.
- Limited in storage capacity, memory capacity, and computing power relative to a PC.
- Generally uses slower processors that consume less battery power and produce less heat.
- The two dominant OSes today are Google Android and Apple iOS.

DISTRIBUTED SYSTEMS

- Distributed Systems consist of multiple, possibly heterogeneous, computers connected together via a network and cooperating in some way, form, or fashion.
- Networks may range from small tight LANs to broad reaching WANs.
    - WAN = Wide Area Network, such as an international corporation
    - MAN =Metropolitan Area Network, covering a region the size of a city for example.
    - LAN =Local Area Network, typical of a home, business, single-site corporation, or university campus.
    - PAN = Personal Area Network, such as the bluetooth connection between your PC, phone, headset, car, etc.
- Network access speeds, throughputs, reliabilities, are all important issues.
- OS view of the network may range from just a special form of file access to complex well-coordinated network operating systems.
- Shared resources may include files, CPU cycles, RAM, printers, and other resources.

CLIENT-SERVER COMPUTING

- A defined server provides services (HW or SW) to other systems which serve as clients. The Figure 1.14 reflects the general structure of a client server system.

- A process may act as both client and server of either the same or different resources.
- Served resources may include disk space, CPU cycles, time of day, IP name information, graphical displays (X Servers), or other resources.

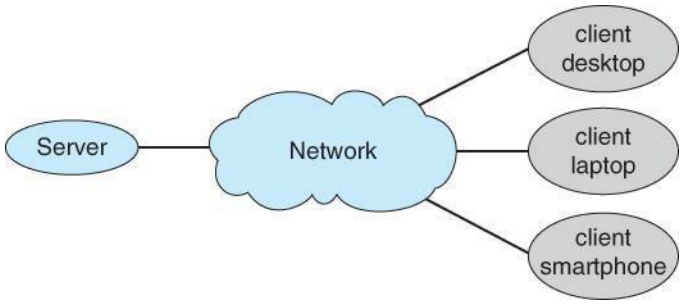


Figure 1.14 - General structure of a client-server system

Peer-to-Peer Computing

- Any computer or process on the network may provide services to any other which requests it. The Figure 1.15 shows the peer to peer computing.
- May employ a central "directory" server for looking up the location of resources, or may use peer-to-peer searching to find resources.
- E.g. Skype uses a central server to locate a desired peer, and then further communication is peer to peer.

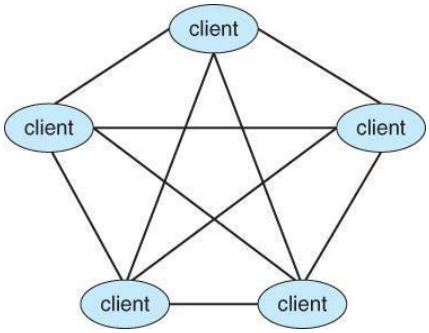


Figure 1.15 - Peer-to-peer system with no centralized service

VIRTUALIZATION

- Allows one or more "guest" operating systems to run on virtual machines hosted by a single physical machine and the virtual machine manager.
- Useful for cross-platform development and support.
- For example, a student could run UNIX on a virtual machine, hosted by a virtual machine manager on a Windows based personal computer. The student would have full root access to the virtual machine, and if it crashed, the underlying Windows machine should be unaffected.

- System calls have to be caught by the VMM and translated into (different) system calls made to the real underlying OS.
- Virtualization can slow down program that have to run through the VMM, but can also speed up some things if virtual hardware can be accessed through a cache instead of a physical device this is shown in Figure 1.16.
- Depending on the implementation, programs can also run simultaneously on the native OS, bypassing the virtual machines.

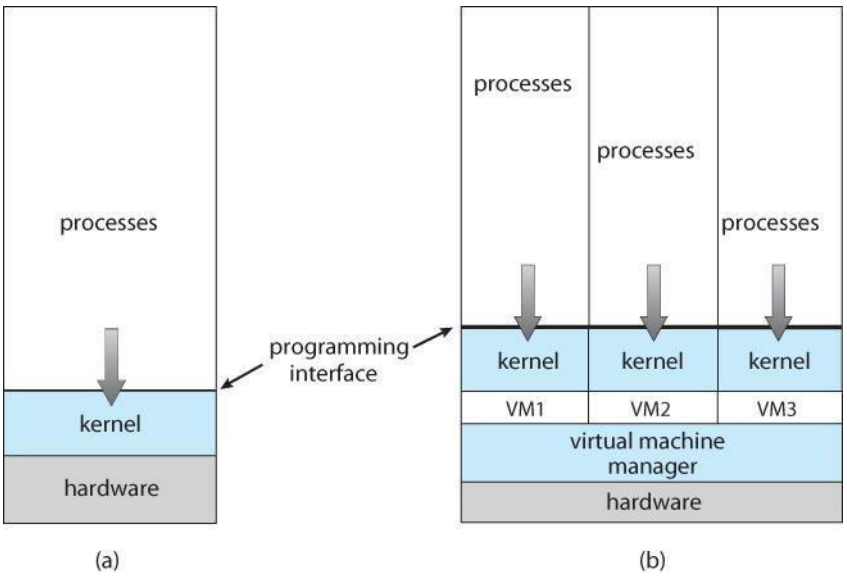


Figure 1.16 - VMWare

CLOUD COMPUTING

- Delivers computing, storage, and applications as a service over a network.
- Types of cloud computing:
    - Public cloud - Available to anyone willing to pay for the service.
    - Private cloud - Run by a company for internal use only.
    - Hybrid cloud - A cloud with both public and private components.
    - **Software as a Service - SaaS** - Applications such as word processors available via the Internet
    - **Platform as a Service - PaaS** - A software stack available for application use, such as a database server
    - **Infrastructure as a Service - IaaS** - Servers or storage available on the Internet, such as backup servers, photo storage, or file storage.
    - Service providers may provide more than one type of service

- Clouds may contain thousands of physical computers, millions of virtual ones, and petabytes of total storage. In Figure 1.17 cloud computing environment is shown.
- Web hosting services may offer (one or more) virtual machine(s) to each of their clients.

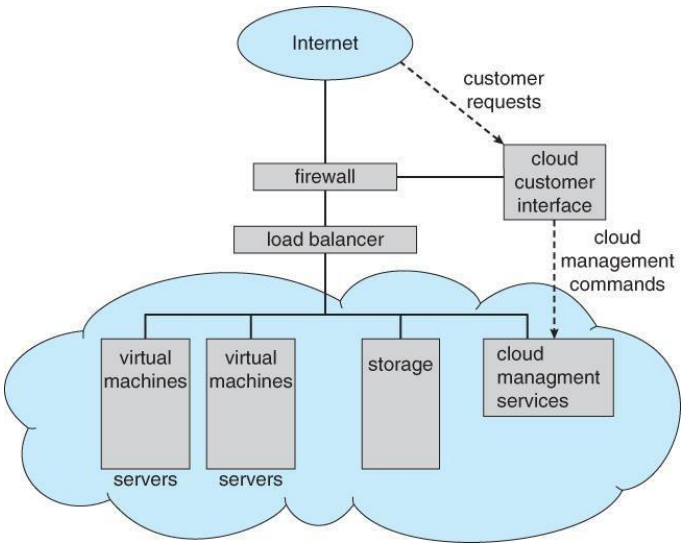


Figure 1.17 - Cloud computing

Real-Time Embedded Systems

- Embedded into devices such as automobiles, climate control systems, process control, and even toasters and refrigerators.
- May involve specialized chips, or generic CPUs applied to a particular task. Process control devices require real-time (interrupt driven) OS. Response time can be critical for many such devices.

2. OPERATING SYSTEM STRUCTURES

An operating system provides the environment within which programs are executed. Internally, operating systems vary greatly in their makeup, since they are organized along many different lines. The design of a new operating system is a major task. It is important that the goals of the system be well defined before the design begins. These goals form the basis for choices among various algorithms and strategies.

SERVICES OF OPERATING SYSTEMS

An operating system provides an environment for the execution of programs. It provides certain services to programs and to the users of those programs. The specific services provided, of course, differ from one operating system to another, but we can identify common classes. These operating system services are provided for the convenience of the

programmer, to make the programming task easier. Figure 2.1 shows one view of the various operating-system services and the communications between them.

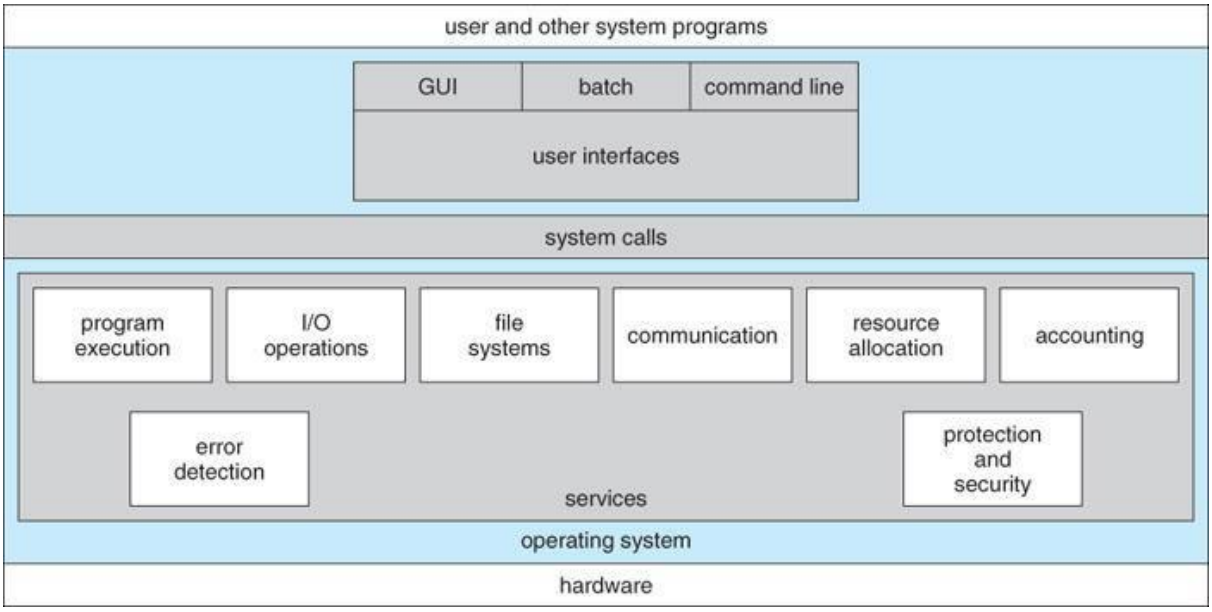


Figure 2.1 A View Of Operating System Services.

User interface.

Almost all operating systems have a **user interface (UI)**. This interface can take several forms. One is a **command-line interface (CLI)**, which uses text commands and a method for entering them (say, a keyboard for typing in commands in a specific format with specific options). Another is a **batch interface**, in which commands and directives to control those commands are entered into files, and those files are executed. Most commonly, a **graphical user interface (GUI)** is used. Here, the interface is a window system with a pointing device to direct I/O, choose from menus, and make selections and a keyboard to enter text. Some systems provide two or all three of these variations.

Program execution.

The system must be able to load a program into memory and to run that program. The program must be able to end its execution, either normally or abnormally (indicating error).

I/O operations.

A running program may require I/O, which may involve a file or an I/O device. For specific devices, special functions may be desired (such as recording to a CD or DVD drive or blanking a display screen). For efficiency and protection, users usually cannot control I/O devices directly. Therefore, the operating system must provide a means to do I/O.

File-system manipulation.

The file system is of particular interest. Obviously, programs need to read and write files and directories. They also need to create and delete them by name, search for a given file, and list file information. Finally, some operating systems include permissions management to allow or deny access to files or directories based on file ownership. Many operating systems provide a variety of file systems, sometimes to allow personal choice and sometimes to provide specific features or performance characteristics.

Communications.

Communication may occur between processes that are executing on the same computer or between processes that are executing on different computer systems tied together by a computer network. Communications may be implemented via **shared memory**, in which two or more processes read and write to a shared section of memory, or **message passing**, in which packets of information in predefined formats are moved between processes by the operating system.

Error detection.

The operating system needs to be detecting and correcting errors constantly. Errors may occur in the CPU and memory hardware (such as a memory error or a power failure), in I/O devices (such as a parity error on disk, a connection failure on a network, or lack of paper in the printer), and in the user program (such as an arithmetic overflow, an attempt to access an illegal memory location, or a too-great use of CPU time). For each type of error, the operating system should take the appropriate action to ensure correct and consistent computing. Sometimes, it has no choice but to halt the system. At other times, it might terminate an error causing process or return an error code to a process for the process to detect and possibly correct. Another set of operating system functions exists not for helping the user but rather for ensuring the efficient operation of the system itself. Systems with multiple users can gain efficiency by sharing the computer resources among the users.

Resource allocation.

When there are multiple users or multiple jobs running at the same time, resources must be allocated to each of them. The operating system manages many different types of resources. Some (such as CPU cycles, main memory, and file storage) may have special allocation code, whereas others (such as I/O devices) may have much more general request and release code. For instance, in determining how best to use the CPU, operating systems have CPU-scheduling routines that take into account the speed of the CPU, the jobs that must

be executed, the number of registers available, and other factors. There may also be routines to allocate printers, USB storage drives, and other peripheral devices.

Accounting.

We want to keep track of which users use how much and what kinds of computer resources. This record keeping may be used for accounting (so that users can be billed) or simply for accumulating usage statistics. Usage statistics may be a valuable tool for researchers who wish to reconfigure the system to improve computing services.

Protection and security.

The owners of information stored in a multiuser or networked computer system may want to control use of that information. When several separate processes execute concurrently, it should not be possible for one process to interfere with the others or with the operating system itself. Protection involves ensuring that all access to system resources is controlled. Security of the system from outsiders is also important. Such security starts with requiring each user to authenticate. If a system is to be protected and secure, precautions must be instituted throughout it. A chain is only as strong as its weakest link.

USER AND OPERATING-SYSTEM INTERFACE

Operating system has two fundamental approaches.

1. **Command-Line Interface or Command Interpreter** that allows users to directly enter commands to be performed by the operating system.
2. Users to interface with the operating system via a **Graphical User Interface or GUI**.

COMMAND INTERPRETER

- Gets and processes the next user request, and launches the requested programs.
- In some systems the CI may be incorporated directly into the kernel.
- More commonly the CI is a separate program that launches once the user logs in or otherwise accesses the system.
- UNIX, for example, provides the user with a choice of different shells, **Bourne shell**, **C shell**, **Bourne-Again shell**, **Korn shell**, and others which may either be configured to launch automatically at login, or which may be changed on the fly. Figure 2.2 shows the Bourne shell command interpreter being used on Solaris 10.
- Different shells provide different functionality, in terms of certain commands that are implemented directly by the shell without launching any external programs. Most

provide at least a rudimentary command interpretation structure for use in shell script programming (loops, decision constructs, variables)

- An interesting distinction is the processing of wild card file naming and I/O re-direction. On UNIX systems those details are handled by the shell, and the program which is launched sees only a list of filenames generated by the shell from the wild cards. On a DOS system, the wild cards are passed along to the programs, which can interpret the wild cards as the program sees fit.

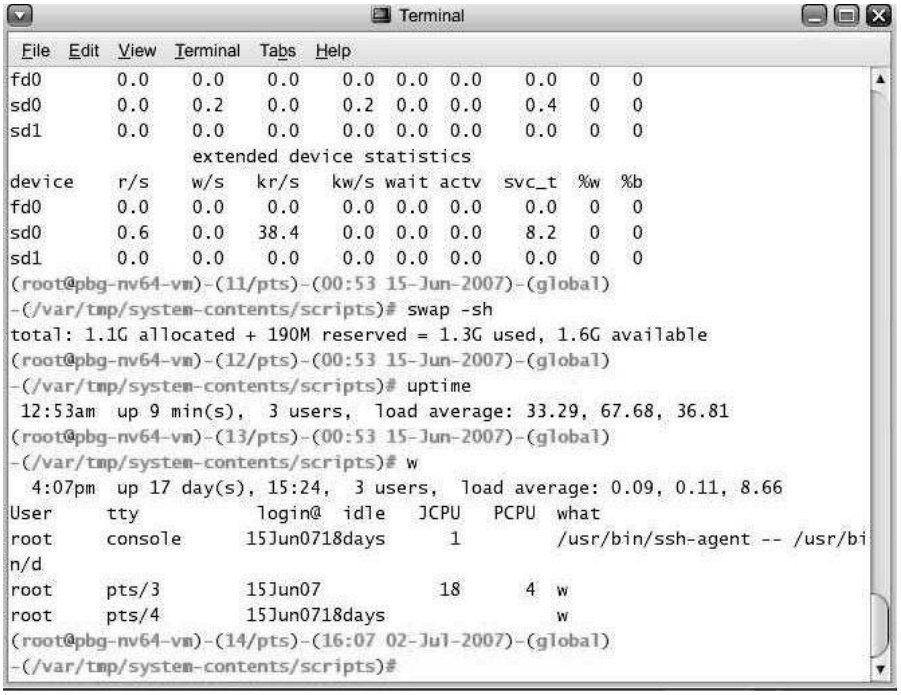


Figure 2.2 The Bourne shell command interpreter in Solrais 10.

GRAPHICAL USER INTERFACE (GUI)

A second strategy for interfacing with the operating system is through a user friendly graphical user interface, or GUI. Here, rather than entering commands directly via a command-line interface, users employ a mouse-based window and- menu system characterized by a **desktop** metaphor. The user moves the mouse to position its pointer on images, or **icons**, on the screen (the desktop) that represent programs, files, directories, and system functions. Depending on the mouse pointer's location, clicking a button on the mouse can invoke a program, select a file or directory—known as a **folder**—or pull down a menu that contains commands.

- Generally implemented as a desktop metaphor, with file folders, trash cans, and resource icons.

- Icons represent some item on the system, and respond accordingly when the icon is activated.
- First developed in the early 1970's at Xerox PARC research facility.
- In some systems the GUI is just a front end for activating a traditional command line interpreter running in the background. In others the GUI is a true graphical shell in its own right.
- Mac has traditionally provided ONLY the GUI interface. With the advent of OSX (based partially on UNIX), a command line interface has also become available.
- Because mice and keyboards are impractical for small mobile devices, these normally use a touch-screen interface today that responds to various patterns of swipes or "gestures". When these first came out they often had a physical keyboard and/or a trackball of some kind built in, but today a virtual keyboard is more commonly implemented on the touch screen.

CHOICE OF INTERFACE

The choice of whether to use a command-line or GUI interface is mostly one of personal preference **System administrators** who manage computers and **power users** who have deep knowledge of a system frequently use the command-line interface. It is more efficient, giving faster access to the activities needed to perform. Indeed, on some systems, only a subset of system functions is available via the GUI, leaving the less common tasks to those who are command-line knowledgeable. Further, command line interfaces usually make repetitive tasks easier, in part because they have their own programmability. For example, if a frequent task requires a set of command-line steps, those steps can be recorded into a file, and that file can be run just like a program. The program is not compiled into executable code but rather is interpreted by the command-line interface. These **shell scripts** are very common on systems that are command-line oriented, such as UNIX and Linux.

The user interface can vary from system to system and even from user to user within a system. It typically is substantially removed from the actual system structure. The design of a useful and friendly user interface is therefore not a direct function of the operating system. In this book, we concentrate on the fundamental problems of providing adequate service to user programs. From the point of view of the operating system, we do not distinguish between user programs and system programs.

SYSTEM CALLS

- System calls provide a means for user or application programs to call upon the services of the operating system.
- Generally written in C or C++, although some are written in assembly for optimal performance.
- Figure 2.3 illustrates the sequence of system calls required to copy a file:

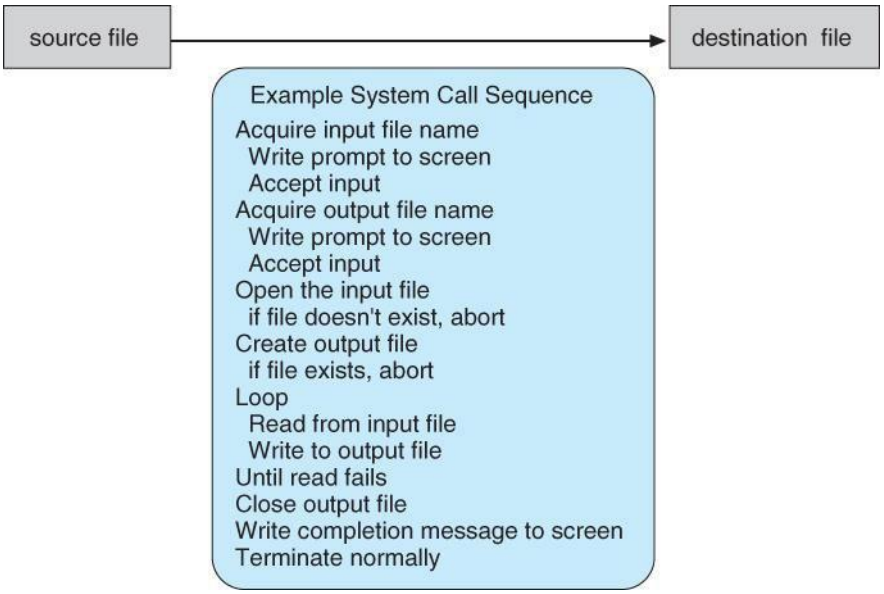


Figure 2.3 Usage of System Calls

As an example of a standard API, consider the read () function that is available in UNIX and Linux systems. The API for this function is obtained from the man page by invoking the command

man read

on the command line. A description of this API appears below:

#include <unistd.h>

ssize_t read(int fd, void *buf, size_t count)

A program that uses the read() function must include the unistd.h header file, as this file defines the ssize t and size t data types (among other things). The parameters passed to read() are as follows:

• int fd—the file descriptor to be read

• void *buf—a buffer where the data will be read into

• size t count—the maximum number of bytes to be read into the

Buffer On a successful read, the number of bytes read is returned. A return value of 0 indicates end of file. If an error occurs, read() returns −1. Figure 2.4 shows the working of the parameters for the system calls as a table.

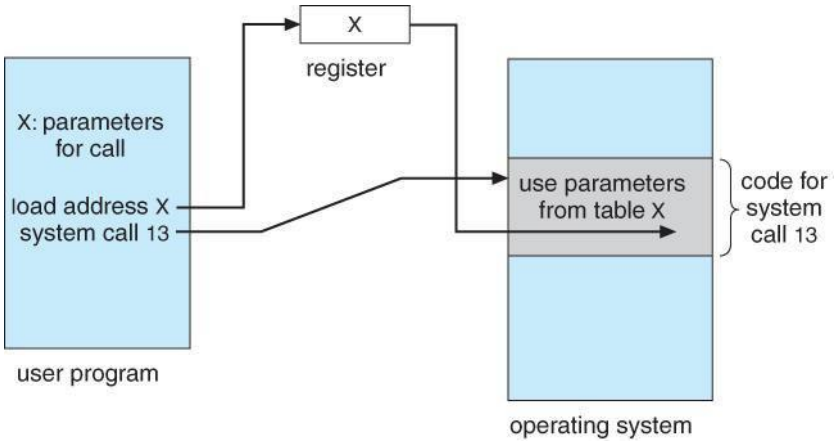


Figure 2.4 - Passing of parameters as a table

TYPES OF SYSTEM CALLS

System calls can be grouped roughly into six major categories: process control, file manipulation, device manipulation, information maintenance, communications, and protection.

Most of the system calls support, or supported by, concepts and functions. Figure 2.5 summarizes the types of system calls normally provided by an operating system. Examples are provided for the actual counterparts to the system calls for Windows, UNIX, and Linux systems.

EXAMPLES OF WINDOWS AND UNIX SYSTEM CALLS

	Windows	Unix
Process Control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File Manipulation	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Device Manipulation	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
Information Maintenance	GetCurrentProcessID() SetTimer() Sleep()	getpid() alarm() sleep()
Communication	CreatePipe() CreateFileMapping() MapViewOfFile()	pipe() shmget() mmap()
Protection	SetFileSecurity() InitlializeSecurityDescriptor() SetSecurityDescriptorGroup()	chmod() umask() chown()

Figure 2.5 Example for UNIX and windows system calls

The example of system calls for UNIX and windows is shown in the Figure 2.6

- Process control
  - end, abort
  - load, execute
  - create process, terminate process
  - get process attributes, set process attributes
  - wait for time
  - wait event, signal event
  - allocate and free memory
- File management
  - create file, delete file
  - open, close
  - read, write, reposition
  - get file attributes, set file attributes
- Device management
  - request device, release device
  - read, write, reposition
  - get device attributes, set device attributes
  - logically attach or detach devices
- Information maintenance
  - get time or date, set time or date
  - get system data, set system data
  - get process, file, or device attributes
  - set process, file, or device attributes
- Communications
  - create, delete communication connection
  - send, receive messages
  - transfer status information
  - attach or detach remote devices

Figure 2.6 Types of System calls

PROCESS CONTROL

A running program needs to be able to halt its execution either normally (end()) or abnormally (abort()). If a system call is made to terminate the currently running program abnormally, or if the program runs into a problem and causes an error trap, a dump of memory is sometimes taken and an error message generated.

The dump is written to disk and may be examined by a debugger—a system program designed to aid the programmer in finding and correcting errors, or bugs—to determine the cause of the problem. Under either normal or abnormal circumstances, the operating system must transfer control to the invoking command interpreter. The command interpreter then reads the next command. In an interactive system, the command interpreter simply continues

with the next command; it is assumed that the user will issue an appropriate command to respond to any error.

- Process control system calls include end, abort, load, execute, create process, terminate process, get/set process attributes, wait for time or event, signal event, and allocate and free memory.
- Processes must be created, launched, monitored, paused, resumed, and eventually stopped.
- When one process pauses or stops, then another must be launched or resumed
- When processes stop abnormally it may be necessary to provide core dumps and/or other diagnostic or recovery tools.
- Compare DOS (a single-tasking system) with UNIX (a multi-tasking system).
    - When a process is launched in DOS, the command interpreter first unloads as much of itself as it can to free up memory, then loads the process and transfers control to it. The interpreter does not resume until the process has completed, as shown in Figure 2.7

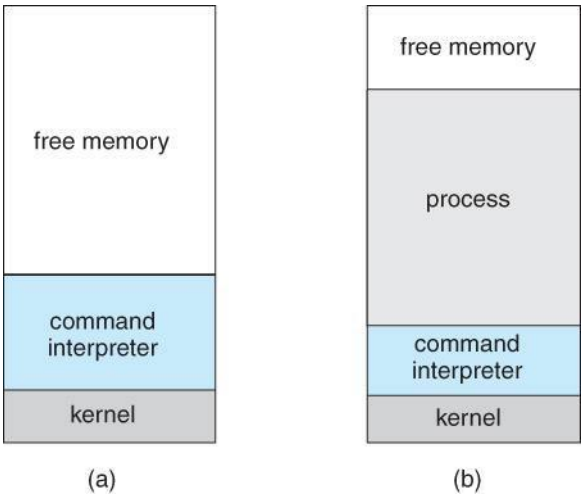


Figure 2.7 - MS-DOS execution. (a) At system start up. (b) Running a program.

    - Because UNIX is a multi-tasking system, the command interpreter remains completely resident when executing a process, as shown in Figure 2.11 below.
- The user can switch back to the command interpreter at any time, and can place the running process in the background even if it was not originally launched as a background process.

- The command interpreter first executes a "fork" system call, which creates a second process which is an exact duplicate (clone) of the original command interpreter. The original process is known as the parent, and the cloned process is known as the child, with its own unique process ID and parent ID.
- The child process then executes an "exec" system call, which replaces its code with that of the desired process. Figure 2.8 shows the FreeBSD running multiple programs.
- The parent (command interpreter) normally waits for the child to complete before issuing a new command prompt, but in some cases it can also issue a new prompt right away, without waiting for the child process to complete. (The child is then said to be running "in the background", or "as a background process".)

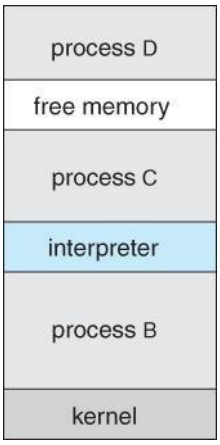


Figure 2.8 - FreeBSD running multiple programs

FILE MANAGEMENT

- File management system calls include create file, delete file, open, close, read, write, reposition, get file attributes, and set file attributes.
- These operations may also be supported for directories as well as ordinary files.
- (The actual directory structure may be implemented using ordinary files on the file system, or through other means.

DEVICE MANAGEMENT

- Device management system calls include request device, release device, read, write, reposition, get/set device attributes, and logically attach or detach devices.
- Devices may be physical (e.g. disk drives), or virtual / abstract (e.g. files, partitions, and RAM disks).

- Some systems represent devices as special files in the file system, so that accessing the "file" calls upon the appropriate device drivers in the OS. See for example the /dev directory on any UNIX system.

INFORMATION MAINTENANCE

- Information maintenance system calls include calls to get/set the time, date, system data, and process, file, or device attributes.
- Systems may also provide the ability to dump memory at any time, single step programs pausing execution after each instruction, and tracing the operation of programs, all of which can help to debug programs.

COMMUNICATION

- Communication system calls create/delete communication connection, send/receive messages, transfer status information, and attach/detach remote devices.
- The **message passing** model must support calls to:
    - Identify a remote process and/or host with which to communicate.
    - Establish a connection between the two processes.
    - Open and close the connection as needed.
    - Transmit messages along the connection.
    - Wait for incoming messages, in either a blocking or non-blocking state.
    - Delete the connection when no longer needed.
- The **shared memory** model must support calls to:
    - Create and access memory that is shared amongst processes and threads.
    - Provide locking mechanisms restricting simultaneous access.
    - Free up shared memory and/or dynamically allocate it as needed.
- Message passing is simpler and easier, (particularly for inter-computer communications), and is generally appropriate for small amounts of data.
- Shared memory is faster, and is generally the better approach where large amounts of data are to be shared, (particularly when most processes are reading the data rather than writing it, or at least when only one or a small number of processes need to change any given data item.)

PROTECTION

- Protection provides mechanisms for controlling which users / processes have access to which system resources.

- System calls allow the access mechanisms to be adjusted as needed, and for non-privileged users to be granted elevated access permissions under carefully controlled temporary circumstances.
- Once only of concern on multi-user systems, protection is now important on all systems, in the age of ubiquitous network connectivity.

SYSTEM PROGRAMS

- System programs provide OS functionality through separate applications, which are not part of the kernel or command interpreters. They are also known as system utilities or system applications.
- Most systems also ship with useful applications such as calculators and simple editors, (e.g. Notepad). Some debate arises as to the border between system and non-system applications.
- System programs may be divided into these categories:
    - **File management** - programs to create, delete, copy, rename, print, list, and generally manipulate files and directories.
    - **Status information** - Utilities to check on the date, time, number of users, processes running, data logging, etc. System **registries** are used to store and recall configuration information for particular applications.
    - **File modification** - e.g. text editors and other tools which can change file contents.
    - **Programming-language support** - E.g. Compilers, linkers, debuggers, profilers, assemblers, library archive management, interpreters for common languages, and support for make.
    - **Program loading and execution** - loaders, dynamic loaders, overlay loaders, etc., as well as interactive debuggers.
    - **Communications** - Programs for providing connectivity between processes and users, including mail, web browsers, remote logins, file transfers, and remote command execution.
    - **Background services** - System daemons are commonly started when the system is booted, and run for as long as the system is running, handling necessary services. Examples include network daemons, print servers, process schedulers, and system error monitoring services.

- Most operating systems today also come complete with a set of **application programs** to provide additional services, such as copying files or checking the time and date.
- Most users' views of the system is determined by their command interpreter and the application programs. Most never make system calls through the API, (with the exception of simple (file) I/O in user-written programs.)

OPERATING-SYSTEM DESIGN AND IMPLEMENTATION

DESIGN GOALS

- **Requirements** define properties which the finished system must have, and are a necessary first step in designing any large complex system.
    - **User requirements** are features that users care about and understand, and are written in commonly understood vernacular. They generally do not include any implementation details, and are written similar to the product description one might find on a sales brochure or the outside of a shrink-wrapped box.
    - **System requirements** are written for the developers, and include more details about implementation specifics, performance requirements, compatibility constraints, standards compliance, etc. These requirements serve as a "contract" between the customer and the developers, (and between developers and subcontractors), and can get quite detailed.
- Requirements for operating systems can vary greatly depending on the planned scope and usage of the system. (Single user / multi-user, specialized system / general purpose, high/low security, performance needs, operating environment, etc.)

MECHANISMS AND POLICIES

- Policies determine *what* is to be done. Mechanisms determine *how* it is to be implemented.
- If properly separated and implemented, policy changes can be easily adjusted without re-writing the code, just by adjusting parameters or possibly loading new data / configuration files. For example the relative priority of background versus foreground tasks.

IMPLEMENTATION

- Traditionally OSes were written in assembly language. This provided direct control over hardware-related issues, but inextricably tied a particular OS to a particular HW platform.
- Recent advances in compiler efficiencies mean that most modern OS are written in C, or more recently, C++. Critical sections of code are still written in assembly language,
- Operating systems may be developed using **emulators** of the target hardware, particularly if the real hardware is unavailable or not a suitable platform for development, (e.g. smart phones, game consoles, or other similar devices.)

OPERATING-SYSTEM STRUCTURE

For efficient performance and implementation an OS should be partitioned into separate subsystems, each with carefully defined tasks, inputs, outputs, and performance characteristics. These subsystems can then be arranged in various architectural configurations:

SIMPLE STRUCTURE

When DOS was originally written its developers had no idea how big and important it would eventually become. It was written by a few programmers in a relatively short amount of time, without the benefit of modern software engineering techniques, and then gradually grew over time to exceed its original expectations. It does not break the system into subsystems, and has no distinction between user and kernel modes, allowing all programs direct access to the underlying hardware. (Note that user versus kernel mode was not supported by the 8088 chip set anyway, so that really wasn't an option back then.)

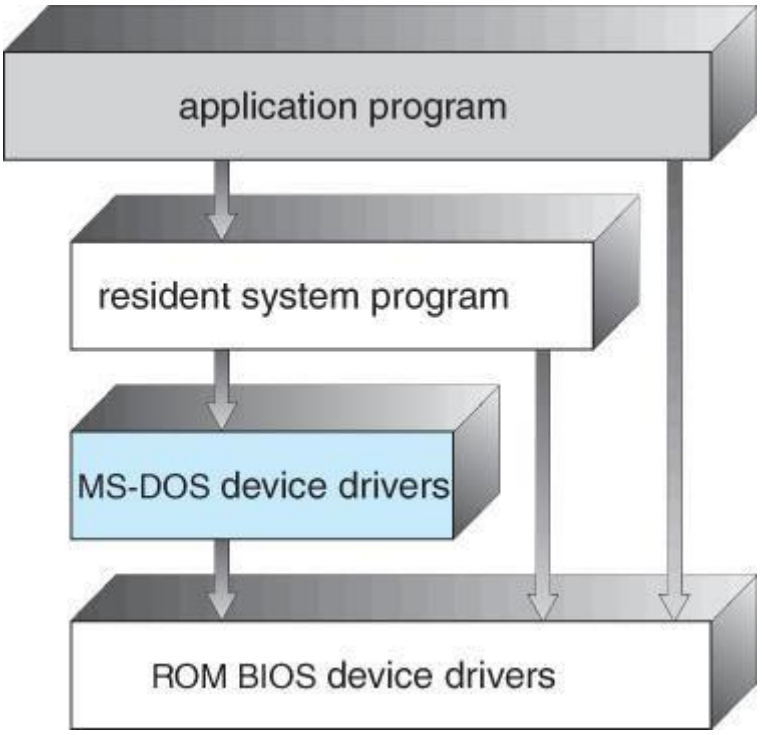


Figure 2.9 - MS-DOS layer structure

The original UNIX OS used a simple layered approach, but almost all the OS was in one big layer, not really breaking the OS down into layered subsystems:

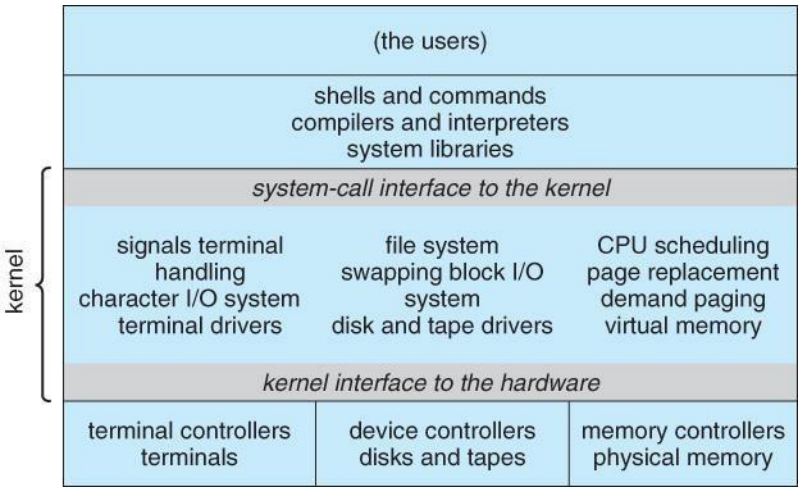


Figure 2.10 - Traditional UNIX system structure

LAYERED APPROACH

- Another approach is to break the OS into a number of smaller layers, each of which rests on the layer below it, and relies solely on the services provided by the next lower layer.

- This approach allows each layer to be developed and debugged independently, with the assumption that all lower layers have already been debugged and are trusted to deliver proper services.
- The problem is deciding what order in which to place the layers, as no layer can call upon the services of any higher layer, and so many chicken-and-egg situations may arise.
- Layered approaches can also be less efficient, as a request for service from a higher layer has to filter through all lower layers before it reaches the HW, possibly with significant processing at each step.
- Figure 2.11 shows the details of the layered operating system.
- **2.7.3 MICROKERNELS**
- The basic idea behind micro kernels is to remove all non-essential services from the kernel, and implement them as system applications instead, thereby making the kernel as small and efficient as possible. Most microkernels provide basic process and memory management, and message passing between other services, and not much more.
- Security and protection can be enhanced, as most services are performed in user mode, not kernel mode.

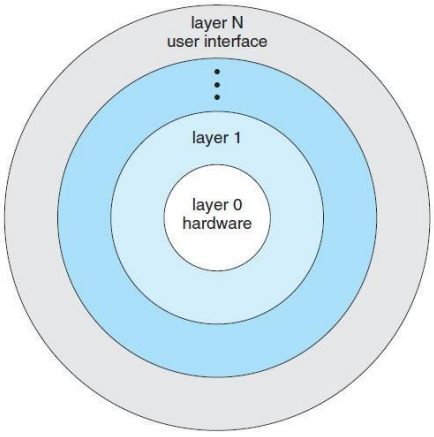


Figure 2.11 - A layered operating system

System expansion can also be easier, because it only involves adding more system applications, not rebuilding a new kernel. Mach was the first and most widely known microkernel, and now forms a major component of Mac OSX.

Windows NT was originally microkernel, but suffered from performance problems relative to Windows 95. NT 4.0 improved performance by moving more services into the kernel, and now XP is back to being more monolithic. Another microkernel example is QNX, a real-time OS for embedded systems.

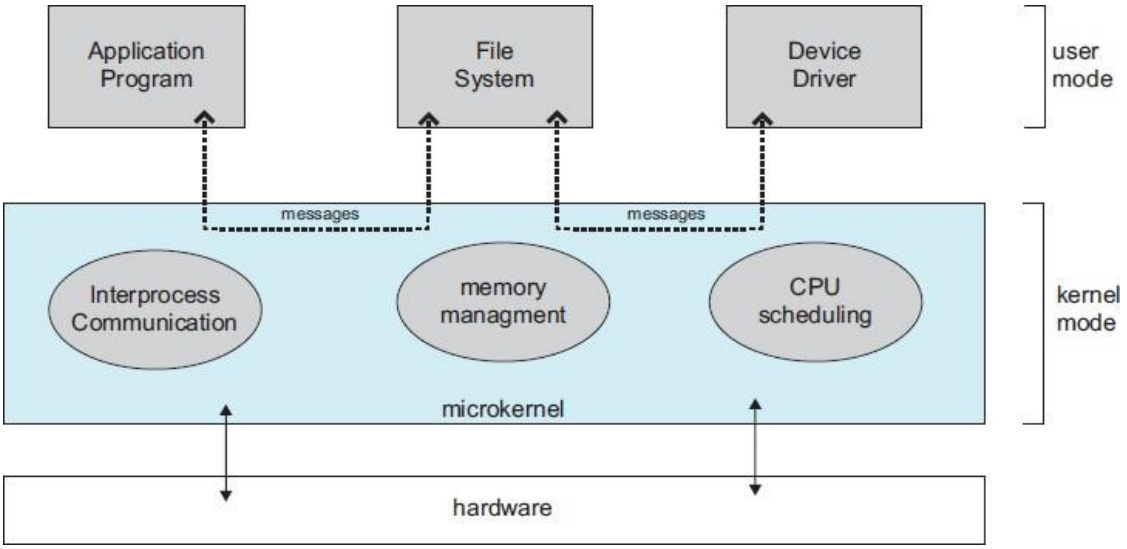


Figure 2.12 - Architecture of a typical microkernel

MODULES

- Modern OS development is object-oriented, with a relatively small core kernel and a set of ***modules*** which can be linked in dynamically. See for example the Solaris structure, as shown in Figure 2.13 below.
- Modules are similar to layers in that each subsystem has clearly defined tasks and interfaces, but any module is free to contact any other module, eliminating the problems of going through multiple intermediary layers, as well as the chicken-and-egg problems.
- The kernel is relatively small in this architecture, similar to microkernels, but the kernel does not have to implement message passing since modules are free to contact each other directly.

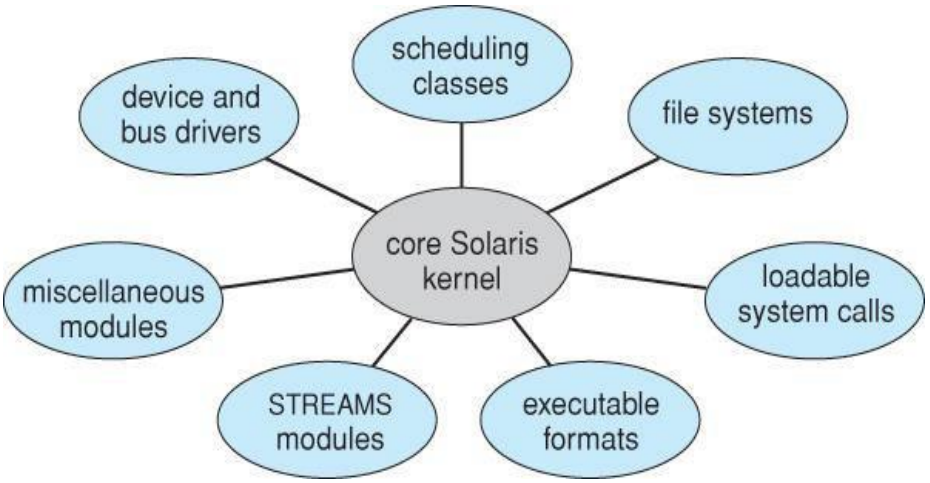


Figure 2.13 Solaris loadable modules

PROCESS MANAGEMENT

A ***process*** can be thought of as a program in execution. A process will need certain resources—such as CPU time, memory, files, and I/O devices —to accomplish its task. These resources are allocated to the process either when it is created or while it is executing. A process is the unit of work in most systems. Systems consist of a collection of processes: operating-system processes execute system code, and user processes execute user code. All these processes may execute concurrently. Although traditionally a process contained only a single *thread* of control as it ran, most modern operating systems now support processes that have multiple threads. The operating system is responsible for several important aspects of process and thread management: the creation and deletion of both user and system processes; the scheduling of processes; and the provision of mechanisms for synchronization, communication, and deadlock handling for processes.

PROCESSES

Firmer control and more compartmentalization of the various programs; and these needs resulted in the notion of a **process**, which is a program in execution. A process is the unit of work in a modern time-sharing system.

A batch system executes **jobs**, whereas a time-shared system has **user programs**, or **tasks**. Even on a single-user system, a user may be able to run several programs at one time: a word processor, a Web browser, and an e-mail package. And even if a user can execute only one program at a time, such as on an embedded device that does not support multitasking, the operating system may need to support its own internal programmed activities, such as memory management.

In many respects, all these activities are similar, so we call all of them **processes**. The terms ***job*** and ***process*** are used almost interchangeably in this text. Although we personally prefer the term ***process***, much of operating-system theory and terminology was developed during a time when the major activity of operating systems was job processing. It would be misleading to avoid the use of commonly accepted terms that include the word ***job*** (such as ***job scheduling***) simply because ***process*** has superseded ***job.***

A process is a program in execution. A process is more than the program code, which is sometimes known as the **text section**. It also includes the current activity, as represented by the value of the **program counter** and the contents of the processor's registers. A process generally also includes the process **stack**, which contains temporary data (such as function parameters, return addresses, and local variables), and a **data section**, which contains global variables. A process may also include a **heap**, which is memory that is dynamically allocated during process run time. The structure of a process in memory is shown in Figure 2.14.

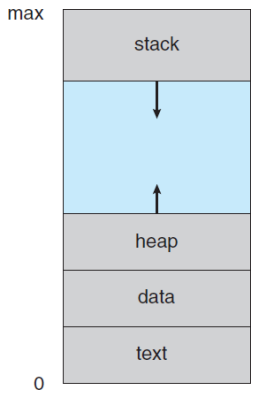


Figure 2.14 Process in memory

PROCESS STATE

- Processes may be in one of 5 states, as shown in Figure 2.15 below.
    - **New** - The process is in the stage of being created.
    - **Ready** - The process has all the resources available that it needs to run, but the CPU is not currently working on this process's instructions.
    - **Running** - The CPU is working on this process's instructions.
    - **Waiting** - The process cannot run at the moment, because it is waiting for some resource to become available or for some event to occur. For example the process may be waiting for keyboard input, disk access request, inter-process messages, a timer to go off, or a child process to finish.

- **Terminated -** The process has completed.

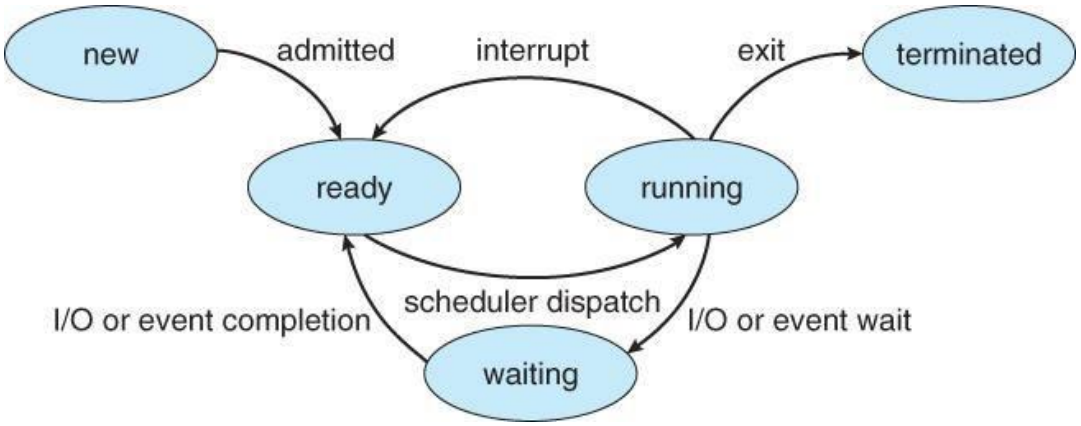


Figure 2.15 - Diagram of process state

- The load average reported by the "w" command indicate the average number of processes in the "Ready" state over the last 1, 5, and 15 minutes, i.e. processes who have everything they need to run but cannot because the CPU is busy doing something else.
- Some systems may have other states besides the ones listed here.

PROCESS CONTROL BLOCK

Each process is represented in the operating system by a **Process Control Block (PCB)**—also called a **task control block**. A PCB is shown in Figure 2.16. It contains many pieces of information associated with a specific process, including these:

• Process state. The state may be new, ready, running, and waiting, halted, and so on.

• Process ID, and parent process ID.

• Program counter. The counter indicates the address of the next instruction to be **executed for this process.**

• CPU registers. The registers vary in number and type, depending on the computer architecture. They include accumulators, index registers, stack pointers, and general-purpose registers, plus any condition-code information. Along with the program counter, this state information must be saved when an interrupt occurs, to allow the process to be continued correctly afterward (Figure 2.17).

• CPU-scheduling information. This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters.

• **Memory-management information.** This information may include such items as the value of the base and limit registers and the page tables, or the segment tables, depending on the memory system used by the operating system.

Figure 2.16 - Process control block (PCB)

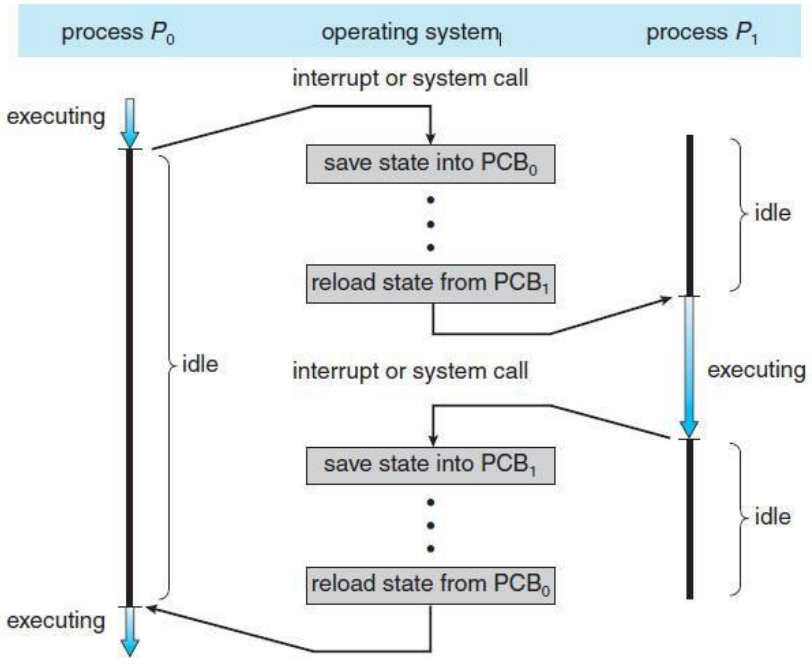


Figure 2.17 Diagram showing CPU switch from process to process.

THREADS

The process model discussed so far has implied that a process is a program that performs a single thread of execution. For example, when a process is running a word processor program, a single thread of instructions is being executed. This single thread of control allows the process to perform only one task at a time. The user cannot simultaneously

type in characters and run the spell checker within the same process, for example. Most modern operating systems have extended the process concept to allow a process to have multiple threads of execution and thus to perform more than one task at a time.

This feature is especially beneficial on multicore systems, where multiple threads can run in parallel. On a system that supports threads, the PCB is expanded to include information for each thread. Other changes throughout the system are also needed to support threads.

PROCESS SCHEDULING

The objective of multiprogramming is to have some process running at all times, to maximize CPU utilization. The objective of time sharing is to switch the CPU among processes so frequently that users can interact with each program, while it is running. To meet these objectives, the **process scheduler** selects an available process (possibly from a set of several available processes) for program execution on the CPU. For a single-processor system, there will never be more than one running process. If there are more processes, the rest will have to wait until the CPU is free and can be rescheduled.

SCHEDULING QUEUES

As processes enter the system, they are put into a **job queue**, which consists of all processes in the system. The processes that are residing in main memory and are ready and waiting to execute are kept on a list called the **ready queue**. This queue is generally stored as a linked list. A ready-queue header contains pointers to the first and final PCBs in the list. Each PCB includes a pointer field that points to the next PCB in the ready queue. The system also includes other queues.

When a process is allocated the CPU, it executes for a while and eventually quits, is interrupted, or waits for the occurrence of a particular event, such as the completion of an I/O request. Suppose the process makes an I/O request to a shared device, such as a disk. Since there are many processes in the system, the disk may be busy with the I/O request of some other process.

The process therefore may have to wait for the disk. The list of processes waiting for a particular I/O device is called a **device queue**. Each device has its own device queue (Figure 2.18).

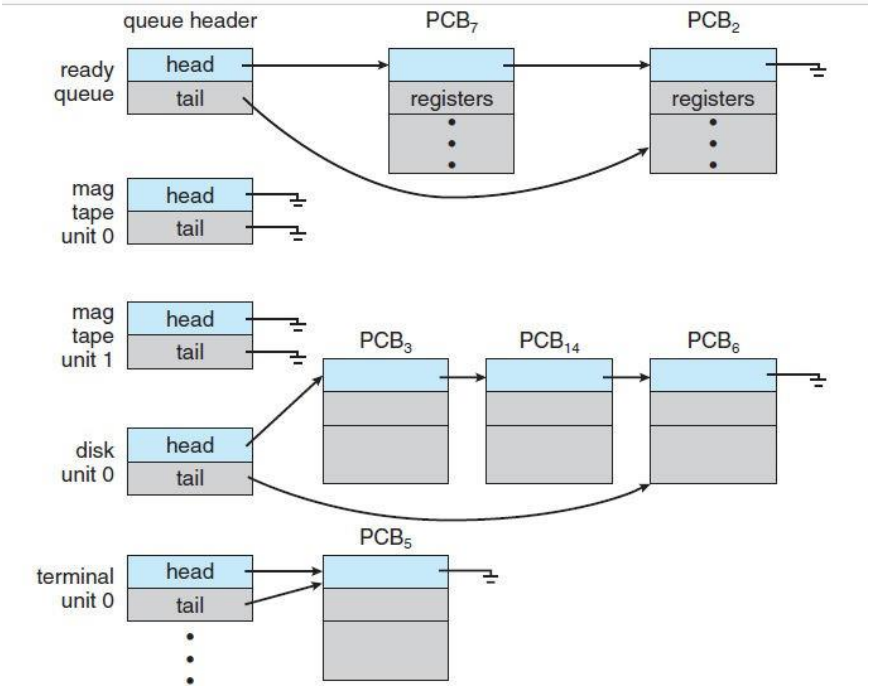


Figure 2.18 The ready queue and various I/O device queues.

A common representation of process scheduling is a queueing diagram, such as that in Figure 2.19. Each rectangular box represents a queue. Two types of queues are present: the ready queue and a set of device queues. The circles represent the resources that serve the queues, and the arrows indicate the flow of processes in the system. A new process is initially put in the ready queue. It waits there until it is selected for execution, or dispatched. Once the process is allocated the CPU and is executing, one of several events could occur:

- The process could issue an I/O request and then be placed in an I/O queue.
- The process could create a new child process and wait for the child's termination.
- The process could be removed forcibly from the CPU, as a result of an interrupt, and be put bac k in the read que ue.

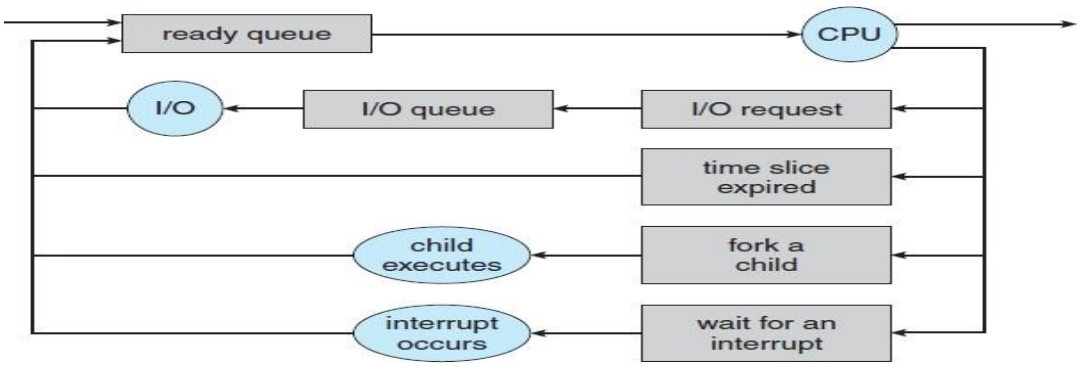


Figure 2.19 Queueing-diagram representation of process scheduling.

SCHEDULERS

A process migrates among the various scheduling queues throughout its lifetime. The operating system must select, for scheduling purposes, processes from these queues in some fashion. The selection process is carried out by the appropriate scheduler. Often, in a batch system, more processes are submitted than can be executed immediately. These processes are spooled to a mass-storage device (typically a disk), where they are kept for later execution. The long-term scheduler, or job scheduler, selects processes from this pool and loads them into memory for execution. The short-term scheduler, or CPU scheduler, selects from among the processes that are ready to execute and allocates the CPU to one of them.

Degree of multiprogramming - The long-term scheduler controls the **degree of multiprogramming** (the number of processes in memory). If the degree of multiprogramming is stable, then the average rate of process creation must be equal to the average departure rate of processes leaving the system.

CPU-bound process – generates I/O requests infrequently, using more of its time doing computations.

I/O-bound process - is one that spends more of its time doing I/O than it spends doing computations.

Medium-term scheduler is diagrammed in Figure 2.20. The key idea behind a medium-term scheduler is that sometimes it can be advantageous to remove a process from memory (and from active contention for the CPU) and thus reduce the degree of multiprogramming.

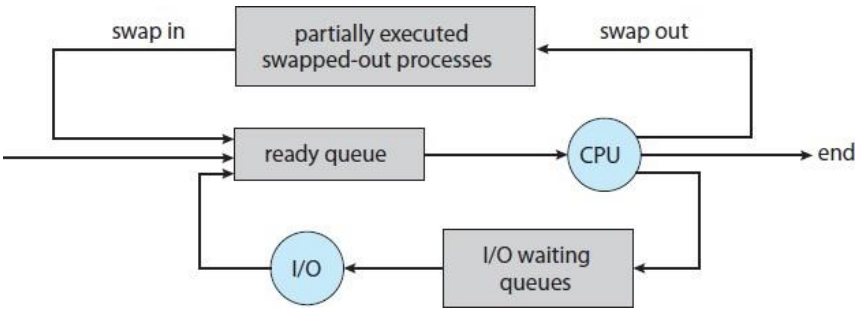


Figure 2.20 Addition of medium-term scheduling to the queueing diagram.

CONTEXT SWITCH

Switching the CPU to another process requires performing a state save of the current process and a state restore of a different process. This task is known as a **Context**

When an interrupt occurs, the system needs to save the current context of the process running on the CPU so that it can restore that context when its processing is done, essentially suspending the process and then resuming it. The context is represented in the PCB of the process.

OPERATIONS ON PROCESSES

PROCESS CREATION

- Processes may create other processes through appropriate system calls, such as **fork** or **spawn**. The process which does the creating is termed the **parent** of the other process, which is termed its **child**.
- Each process is given an integer identifier, termed its **process identifier**, or PID. The parent PID (PPID) is also stored for each process.
- On typical UNIX systems the process scheduler is termed **sched**, and is given PID 0. The first thing it does at system start up time is to launch **init**, which gives that process PID 1. Init then launches all system daemons and user logins, and becomes the ultimate parent of all other processes. Figure 2.21 shows a typical process tree for a Linux system, and other systems will have similar though not identical trees:

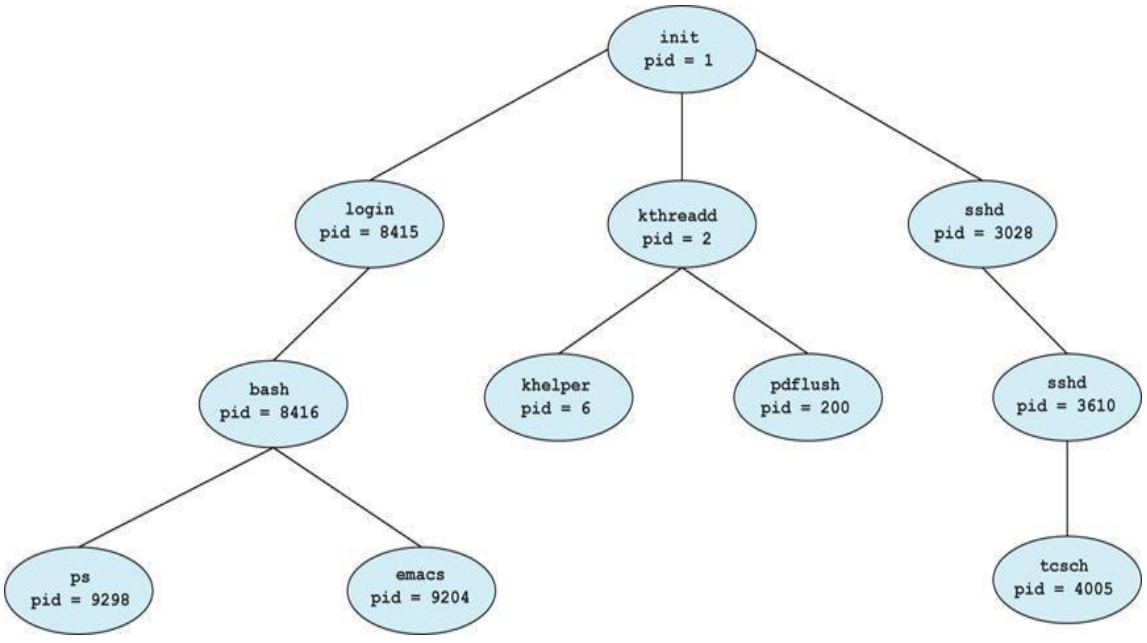


Figure 2.21 - A tree of processes on a typical Linux system

- Depending on system implementation, a child process may receive some amount of shared resources with its parent. Child processes may or may not be limited to a

subset of the resources originally allocated to the parent, preventing runaway children from consuming all of a certain system resource.

- There are two options for the parent process after creating the child:
    1. Wait for the child process to terminate before proceeding. The parent makes a wait( ) system call, for either a specific child or for any child, which causes the parent process to block until the wait( ) returns. UNIX shells normally wait for their children to complete before issuing a new prompt.
    2. Run concurrently with the child, continuing to process without waiting. This is the operation seen when a UNIX shell runs a process as a background task. It is also possible for the parent to run for a while, and then wait for the child later, which might occur in a sort of a parallel processing operation.
- Two possibilities for the address space of the child relative to the parent:
    1. The child may be an exact duplicate of the parent, sharing the same program and data segments in memory. Each will have their own PCB, including program counter, registers, and PID. This is the behaviour of the **fork** system call in UNIX.
    2. The child process may have a new program loaded into its address space, with all new code and data segments. This is the behaviour of the **spawn** system calls in Windows. UNIX systems implement this as a second step, using the **exec** system call.
- Figures 2.22 and 2.23 below shows the fork and exec process on a UNIX system. Note that the fork system call returns the PID of the processes child to each process - It returns a zero to the child process and a non-zero child PID to the parent, so the return value indicates which process is which. Process IDs can be looked up any time for the current process or its direct parent using the getpid( ) and getppid( ) system calls respectively.

```c
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t pid;

   /* fork a child process */
   pid = fork();

   if (pid < 0) { /* error occurred */
      fprintf(stderr, "Fork Failed");
      return 1;
   }
   else if (pid == 0) { /* child process */
      execlp("/bin/ls","ls",NULL);
   }
   else { /* parent process */
      /* parent will wait for the child to complete */
      wait(NULL);
      printf("Child Complete");
   }

   return 0;
}
```

Figure 2.22 Creating a separate process using the UNIX fork() system call.

Figure 2.23 shows the more complicated process for Windows, which must provide all of the parameter information for the new process as part of the forking process.

PROCESS TERMINATION

- Processes may request their own termination by making the exit( ) system call, typically returning an int. This int is passed along to the parent if it is doing a wait( ), and is typically zero on successful completion and some non-zero code in the event of problems.

  child code:

  int exitCode;

  exit(exitCode); // return exitCode; has the same effect when executed from main( )

  parent code:

  pid_t pid;

  int status

  pid = wait(&status);

  // pid indicates which child exited. Exit Code in low-order bits of status

  // macros can test the high-order bits of status for why it stopped

- Processes may also be terminated by the system for a variety of reasons, including:
  - The inability of the system to deliver necessary system resources.
  - In response to a KILL command, or other unhandled process interrupt.
  - A parent may kill its children if the task assigned to them is no longer needed.

- If the parent exits, the system may or may not allow the child to continue without a parent. (On UNIX systems, orphaned processes are generally inherited by init, which then proceeds to kill them. The UNIX *nohup* command allows a child to continue executing after its parent has exited.)

- When a process terminates, all of its system resources are freed up, open files flushed and closed, etc. The process termination status and execution times are returned to the parent if the parent is waiting for the child to terminate, or eventually returned to init if the process becomes an orphan. (Processes which are trying to terminate but which cannot because their parent is not waiting for them are termed zombies. These are eventually inherited by init as orphans and killed off. Note that modern UNIX shells do not produce as many orphans and zombies as older systems used to.)

INTERPROCESS COMMUNICATION

- Independent Processes operating concurrently on a systems are those that can neither affect other processes or be affected by other processes.

- Cooperating Processes are those that can affect or be affected by other processes. There are several reasons why cooperating processes are allowed:

    - Information Sharing - There may be several processes which need access to the same file for example. (e.g. pipelines.)

    - Computation speedup - Often a solution to a problem can be solved faster if the problem can be broken down into sub-tasks to be solved simultaneously (particularly when multiple processors are involved.)

    - Modularity - The most efficient architecture may be to break a system down into cooperating modules. (E.g. databases with a client-server architecture.)

    - Convenience - Even a single user may be multi-tasking, such as editing, compiling, printing, and running the same code in different windows.

```
#include <stdio.h>
#include <windows.h>

int main(VOID)
{
STARTUPINFO si;
PROCESS_INFORMATION pi;

   // allocate memory
   ZeroMemory(&si, sizeof(si));
   si.cb = sizeof(si);
   ZeroMemory(&pi, sizeof(pi));

   // create child process
   if (!CreateProcess(NULL, // use command line
    "C:\\WINDOWS\\system32\\mspaint.exe", // command line
    NULL, // don't inherit process handle
    NULL, // don't inherit thread handle
    FALSE, // disable handle inheritance
    0, // no creation flags
    NULL, // use parent's environment block
    NULL, // use parent's existing directory
    &si,
    &pi))
   {
     fprintf(stderr, "Create Process Failed");
     return -1;
   }
   // parent will wait for the child to complete
   WaitForSingleObject(pi.hProcess, INFINITE);
   printf("Child Complete");

   // close handles
   CloseHandle(pi.hProcess);
   CloseHandle(pi.hThread);
}
```

Figure 2.23 Creating a separate process using Win32 API

- Cooperating processes require some type of inter-process communication, which is most commonly one of two types: Shared Memory systems or Message Passing systems. Figure 2.24 illustrates the difference between the two systems:

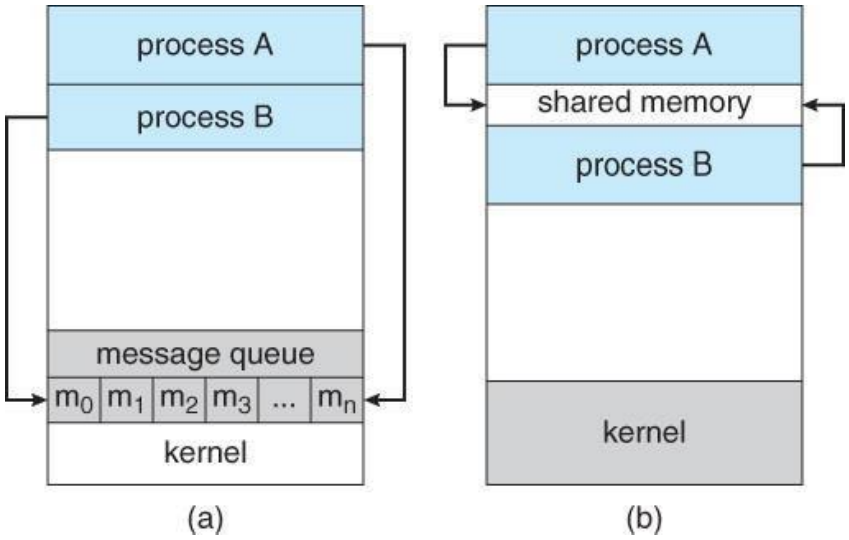


Figure 2.24 - Communications models: (a) Message passing. (b) Shared memory.

- Shared Memory is faster once it is set up, because no system calls are required and access occurs at normal memory speeds. However it is more complicated to set up, and doesn't work as well across multiple computers. Shared memory is generally preferable when large amounts of information must be shared quickly on the same computer.
- Message Passing requires system calls for every message transfer, and is therefore slower, but it is simpler to set up and works well across multiple computers. Message passing is generally preferable when the amount and/or frequency of data transfers is small, or when multiple computers are involved.

Shared-Memory Systems

- In general the memory to be shared in a shared-memory system is initially within the address space of a particular process, which needs to make system calls in order to make that memory publicly available to one or more other processes.
- Other processes which wish to use the shared memory must then make their own system calls to attach the shared memory area onto their address space.
- Generally a few messages must be passed back and forth between the cooperating processes first in order to set up and coordinate the shared memory access.

Producer-Consumer Example Using Shared Memory

This is a classic example, in which one process is producing data and another process is consuming the data. The data is passed via an intermediary buffer, which may be either unbounded or bounded. With a bounded buffer the producer may have to wait until there is space available in the buffer, but with an unbounded buffer the producer will never need to wait. The consumer may need to wait in either case until there is data available.

This example uses shared memory and a circular queue. Note in the code below that only the producer changes "in", and only the consumer changes "out", and that they can never be accessing the same array location at the same time.

First the following data is set up in the shared memory area:

```
#define BUFFER_SIZE 10
typedef struct {
    . . .
} item;
item buffer[ BUFFER_SIZE ];
int in = 0;
int out = 0;
```

- Then the producer process. Note that the buffer is full when "in" is one less than "out" in a circular sense:

```
item nextProduced;
while( true ) {
/* Produce an item and store it in nextProduced */
nextProduced = makeNewItem( . . . );
/* Wait for space to become available */
while( ( ( in + 1 ) % BUFFER_SIZE ) == out )
    ; /* Do nothing */
/* And then store the item and repeat the loop. */
buffer[ in ] = nextProduced;
in = ( in + 1 ) % BUFFER_SIZE;
```

- Then the consumer process. Note that the buffer is empty when "in" is equal to "out":

```
item nextConsumed;
while( true ) {
/* Wait for an item to become available */
while( in == out )
    ; /* Do nothing */
/* Get the next available item */
nextConsumed = buffer[ out ];
out = ( out + 1 ) % BUFFER_SIZE;
/* Consume the item in nextConsumed
   ( Do something with it ) */
}
```

MESSAGE-PASSING SYSTEMS

- Message passing systems must support at a minimum system calls for "send message" and "receive message".
- A communication link must be established between the cooperating processes before messages can be sent.
- There are three key issues to be resolved in message passing systems as further explored in the next three subsections:
    - Direct or indirect communication (naming)
    - Synchronous or asynchronous communication
    - Automatic or explicit buffering.

Naming

- With **direct communication** the sender must know the name of the receiver to which it wishes to send a message.
    - There is a one-to-one link between every sender-receiver pair.
    - For **symmetric** communication, the receiver must also know the specific name of the sender from which it wishes to receive messages. For **asymmetric** communications, this is not necessary.
- **Indirect communication** uses shared mailboxes, or ports.
    - Multiple processes can share the same mailbox or boxes.

- Only one process can read any given message in a mailbox. Initially the process that creates the mailbox is the owner, and is the only one allowed to read mail in the mailbox, although this privilege may be transferred.
- The OS must provide system calls to create and delete mailboxes, and to send and receive messages to/from mailboxes.

SYNCHRONIZATION

- Either the sending or receiving of messages (or neither or both) may be either **blocking** or **non-blocking**. Blocking is considered synchronous Non-blocking is considered asynchronous
- Blocking send has the sender block until the message is received o Blocking receive has the receiver block until a message is available.
- Non-blocking send has the sender send the message and continue o Non-blocking receive has the receiver receive a valid message or null

BUFFERING

- Messages are passed via queues, which may have one of three capacity configurations:
  1. **Zero capacity** - Messages cannot be stored in the queue, so senders must block until receivers accept the messages.
  2. **Bounded capacity**- There is a certain pre-determined finite capacity in the queue. Senders must block if the queue is full, until space becomes available in the queue, but may be either blocking or non-blocking otherwise.
  3. **Unbounded capacity** - The queue has a theoretical infinite capacity, so senders are never forced to block.

```
message next_produced;

while (true) {
     /* produce an item in next_produced */

     send(next_produced);
}
```

Figure 2.25 The Producer Process using message passing

```
message next_consumed;

while (true) {
     receive(next_consumed);

     /* consume the item in next_consumed */
}
```

Figure 2.26 The Producer Process using message passing

COMMUNICATION IN CLIENT-SERVER SYSTEMS

SOCKETS

- A **socket** is an endpoint for communication.
- Two processes communicating over a network often use a pair of connected sockets as a communication channel. Software that is designed for client-server operation may also use sockets for communication between two processes running on the same computer - For example the UI for a database program may communicate with the back-end database manager using sockets. (If the program were developed this way from the beginning, it makes it very easy to port it from a single-computer system to a networked application.)
- A socket is identified by an IP address concatenated with a port number, e.g. 200.100.50.5:80.

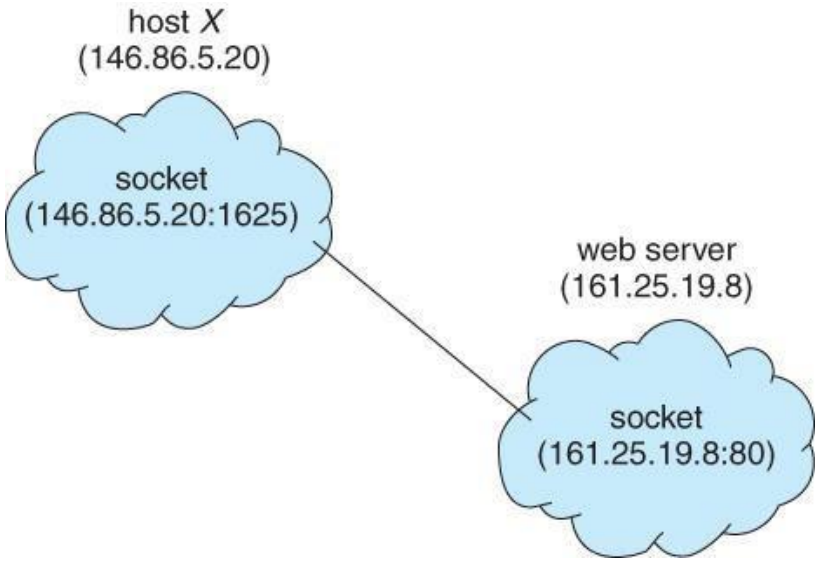


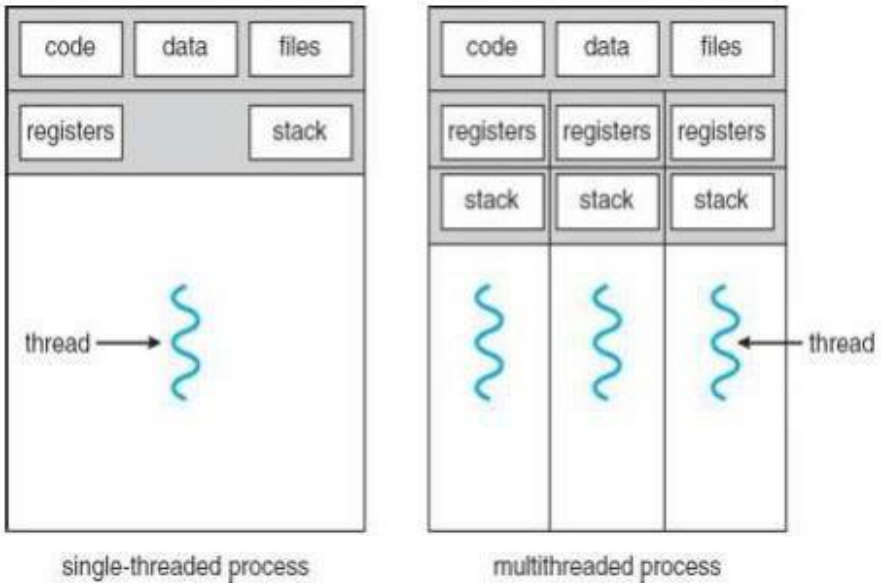
Figure 2.27 communications using sockets

- Port numbers below 1024 are considered to be *well-known*, and are generally reserved for common Internet services. For example, telnet servers listen to port 23, ftp servers to port 21, and web servers to port 80.
- General purpose user sockets are assigned unused ports over 1024 by the operating system in response to system calls such as socket( ) or soctkepair( ).
- Communication channels via sockets may be of one of two major forms:
    - Connection-oriented (TCP, Transmission Control Protocol) connections emulate a telephone connection. All packets sent down the connection are guaranteed to arrive in good condition at the other end, and to be delivered to the receiving process in the order in which they were sent. The TCP layer of the network protocol takes steps to verify all packets sent, re-send packets if necessary, and arrange the received packets in the proper order before delivering them to the receiving process. There is a certain amount of overhead involved in this procedure, and if one packet is missing or delayed, then any packets which follow will have to wait until the errant packet is delivered before they can continue their journey.
    - Connectionless (UDP, User Datagram Protocol) emulate individual telegrams. There is no guarantee that any particular packet will get through undamaged (or at all), and no guarantee that the packets will get delivered in any particular order. There may even be duplicate packets delivered, depending on how the intermediary connections are configured. UDP transmissions are much faster than TCP, but applications must implement their own error checking and recovery procedures.
- Sockets are considered a low-level communications channel, and processes may often choose to use something at a higher level, such as those covered in the next two sections.

THREAD

- A thread is a flow of execution through the process code, with its own program counter, system registers and stack.
- A thread is also called a light weight 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.



2.28 Single threaded vs multithreaded process

User Threads

- Thread management done by user-level threads library
- Three primary thread libraries:
    - POSIX Pthreads
    - Win32 threads
    - Java threads

Kernel Thread

- Supported by the Kernel Examples
    - Windows XP/2000
    - Solaris
    - Linux
    - Tru64 UNIX
    - Mac OS X

Process	Thread
Process is heavy weight or resource intensive.	Thread is light weight taking lesser resources than a process.
Process switching needs interaction with operating system.	Thread switching does not need to interact with operating system.
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.
If one process is blocked then no other process can execute until the first process is unblocked.	While one thread is blocked and waiting, second thread in the same task can run.
Multiple processes without using threads use more resources.	Multiple threaded processes use fewer resources.
In multiple processes each process operates independently of the others.	One thread can read, write or change another thread's data.

Figure 2.29 Difference between Process and thread

THREAD SCHEDULING

- Distinction between user-level and kernel-level threads
- Many-to-one and many-to-many models, thread library schedules user-level threads to run on LWP Known as process-contention scope (PCS) since scheduling competition is within the process
- Kernel thread scheduled onto available CPU is system-contention scope (SCS) – competition among all threads in system.

THREAD POOLS

- Create a number of threads in a pool where they await work
- Advantages:
    - Usually slightly faster to service a request with an existing thread than create a new thread
    - Allows the number of threads in the application(s) to be bound to the size of the pool

TWO MARKS QUESTIONS WITH ANSWERS

1. **What is an Operating system?**

   Ans: An operating system is a program that manages the computer hardware. It also provides a basis for application programs and act as an intermediary between a user of a computer and the computer hardware. It controls and coordinates the use of the hardware among the various application programs for the various users.

2. **What are the objectives of operating system?**

   Ans: An operating system is a program that manages the computer hardware, it act as an intermediate between a user of a computer and the computer hardware. It controls and coordinates the use of the hardware among the various application programs for the various users.

3. **What is the purpose of system programs/system calls?**

   Ans: System programs can be thought of as bundles of useful system calls. They provide basic functionality to users so that users do not need to write their own programs to solve common problems.

4. **Defend timesharing differ from multiprogramming? If so, how?**

   Ans: Main difference between multiprogramming and time sharing is that multiprogramming is the effective utilization of CPU time, by allowing several programs to use the CPU at the same time but time sharing is the sharing of a computing facility by several users that want to use the same facility at the same time.

5. **Compare and contrast DMA and cache memory.**

   Ans: DMA(Direct Memory Access): Direct memory access (DMA) is a feature of computer systems that allows certain hardware subsystems to access main memory (Random-access memory), independent of the central processing unit (CPU). Cache Memory: A cache is a smaller, faster memory, closer to a processor core, which stores copies of the data from frequently used main memory locations. So, both DMA and cache are used for increasing the speed of memory access.

6. **What do you mean by system calls?**

   Ans: System calls provide the interface between a process and the operating system. When a system call is executed, it is treated as by the hardware as software interrupt.

7. **Define process.**

Ans: A process is a program in execution. It is an active entity and it includes the process stack, containing temporary data and the data section contains global variables.

8. **What is process control block?**

   Ans: Each process is represented in the OS by a process control block. It contain many pieces of information associated with a specific process.

9. **What is meant by context switch?**

   Ans: Switching the CPU to another process requires saving the state of the old process and loading the saved state for the new process. This task is known as context switch.

10. **Discuss the difference between symmetric and asymmetric multiprocessing**

    Ans: Symmetric multiprocessing (SMP), in which each processor runs an identical copy of the operating system and these copies, communicate with one another as needed. Asymmetric multiprocessing, in which each processor is assigned a specific task. The master processor controls the system; the other processor looks the master.

11. **What is the main advantage of multiprogramming?**

    Ans: Multiprogramming makes efficient use of the CPU by overlapping the demands for the CPU and its I/O devices from various users. It attempts to increase CPU utilization by always having something for the CPU to execute.

12. **Discuss the main advantages of layered approach to system design?**

    Ans: As in all cases of modular design, designing an operating system in a modular way has several advantages. The system is easier to debug and modify because changes affect only limited sections of the system rather than touching all sections of the operating system. Information is kept only where it is needed and is accessible only within a defined and restricted area, so any bugs affecting that data must be limited to a specific module or layer.

13. **Define inter process communication.**

    Ans: Inter process communication provides a mechanism to allow the co-operating process to communicate with each other and synchronies their actions without sharing the same address space. It is provided a message passing system.

14. **What is bootstrap program?**

Ans: A bootstrap is the program that initializes the operating system (OS) during startup.

15. **Summarize the functions of DMA.**

    Ans: Direct memory access (DMA) is a method that allows an input/output (I/O) device to send or receive data directly to or from the main memory, bypassing the CPU to speed up memory operations. The process is managed by a chip known as a DMA controller (DMAC).

16. **Define: Clustered systems.**

    Ans: A computer cluster is a set of loosely or tightly connected computers that work together so that, in many respects, they can be viewed as a single system.

17. **What is the Kernel?**

    Ans: A more common definition is that the OS is the one program running at all times on the computer, usually called the kernel, with all else being application programs.

18. **List the services provided by an Operating System?**

    Ans: * Program execution
    * I/O Operation
    * File-System manipulation
    * Communications
    * Error detection

19. **What is meant by Batch Systems?**

    Ans: Operators batched together jobs with similar needs and ran through the computer as a group .The operators would sort programs into batches with similar requirements and as system become available, it would run each batch.

20. **What is meant by Time-sharing Systems?**

    Ans: Time Sharing is a logical extension of multiprogramming .Here, CPU executes multiple jobs by switching among them, but the switches occur so frequently that the users can interact with each program while it is running.

5 MARK QUESTIONS

1. What are the various objectives and functions of Operating systems?
2. What are the major activities of an operating systems with regard to process management?
3. Differentiate distributed systems from multiprocessor system?

4. Explain the basic instruction cycle with appropriate diagram?
5. Explain OS structure?
6. Briefly explain virtual machines?
7. Explain about multiprogramming and time sharing operating system?
8. Explain computer system architecture?
9. Explain about system calls?
10. What is OS user interface?

10 MARK QUESTIONS

1. What is system calls in OS? Explain in detail with its types.
2. Discuss the Simple Operating System Structure. Describe the layered approach
3. What are different types of operating system? Explain them in detail
4. Explain User Operating-System Interface in detail
5. Explain operating system functions and services in detail

KEY TERMS

- **Application Programming Interface (API)—**Specification that allows applications to request services from the kernel by making system calls.
- **Client**—Process that requests a service from another process (a server). The machine on which the client process runs is also called a client.
- **Degree of Multiprogramming**—Number of programs a system can manage at a time.
- **Efficient Operating System**—operating system that exhibits high throughput and small turnaround time.
- **Disk Scheduler**—operating system component that determines the order in which disk I/O requests are serviced to improve performance.
- **Distributed Computing**—using multiple independent computers to perform a common task.
- **Distributed Operating System**—Single operating system that provides transparent access to resources spread over multiple computers.

- **Graphical User Interface (GUI)**—User-friendly point of access to an operating system that uses graphical symbols such as windows, icons and menus to facilitate program and file manipulation.
- **Interprocess Communication (IPC) manager**—Operating system component that governs communication between processes.
- **Kernel**—Software that contains the core components of an operating system.
- **Layered Operating System**—Modular operating system that places similar components in isolated layers. Each layer accesses the services of the layer below and returns results to the layer above.
- **Microkernel Operating System**—Scalable operating system that puts a minimal number of services in the kernel and requires user-level programs to implement services generally delegated to the kernel in other types of operating systems.
- **Multiprogramming**—Ability to store multiple programs in memory at once so that they can be executed concurrently.
- **Process Scheduler**—Operating system component that determines which process can gain access to a processor and for how long.
- **Process**—an executing program.
- **Processor-Bound**—Process (or job) that consumes its quantum when executing. These processes (or jobs) tend to be calculation intensive and issue few, if any, I/O requests.
- **System Call**— Call from a user process that invokes a service of the kernel.
- **Thread**—Entity that describes an independently executable stream of program instructions (also called a thread of execution or thread of control). Threads facilitate parallel execution of concurrent activities within a process.
- **Throughput**—Amount of work performed per unit time.
- **Turnaround Time**—Time it takes from the submission of a request until the system returns the result.

UNIT 2. CPU SCHEDULING

INTRODUCTION

CPU scheduling is executed y operating system. Only one process can be acuire CPU at a time of execution, another process will be on hold due to unavailability of resource. It is the duty of operating system to make use of CPU utilization at the most in a multi programmed environment. CPU scheduling is to make the system efficient, fast and fair.

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

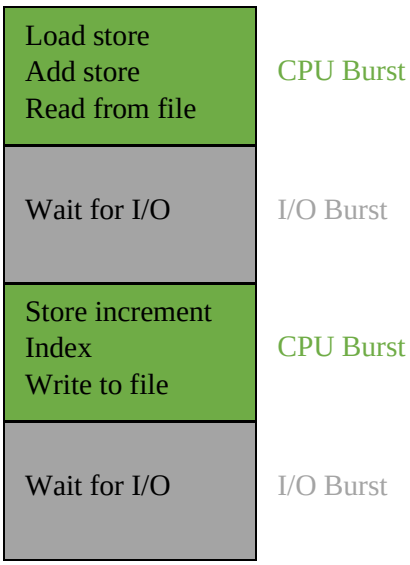


Figure: 2.1 Alternating sequence of CPU

In multiprogramming system, several processes are kept in memory at one time. When one process has to wait, the operating system takes the CPU away from that process and gives the CPU to another process. This sequence continues, every time one process has to wait, another process can take over use of the CPU.

CPU SCHEDULER

Whenever the CPU becomes idle, the operating system must select one of the processes in the ready queue to be executed. The selection process is carried out by the short-

term scheduler (or CPU scheduler). The scheduler selects from among the processes in memory that are ready to execute, and allocates the CPU to one of them. Selection of process by CPU follows the scheduling algorithm.

CPU scheduling decisions may take place when a process:

1. Switches from running to waiting state
2. Switches from running to ready state
3. Switches from waiting to ready
4. Terminates

Scheduling under 1 and 4 is non preemptive, All other scheduling is preemptive

Preemptive

When a process switches from running to a waiting state (due to unavailability of I/O) or terminates

Non preemptive

Once the resource allocated to a process, the process holds the CPU till it gets terminated or it reaches a waiting state

SCHEDULING CRITERIA

There are many different criteria's to check when considering the **"best"** scheduling algorithm, they are:

CPU Utilization

To make out the best use of CPU and not to waste any CPU cycle, CPU would be working most of the time(Ideally 100% of the time). Considering a real system, CPU usage should range from 40% (lightly loaded) to 90% (heavily loaded.)

Throughput

It is the total number of processes completed per unit time or rather say total amount of work done in a unit of time. This may range from 10/second to 1/hour depending on the specific processes.

urnaround Time

It is the amount of time taken to execute a particular process, i.e. The interval from time of submission of the process to the time of completion of the process(Wall clock time).

Waiting Time

The sum of the periods spent waiting in the ready queue amount of time a process has been waiting in the ready queue to acquire get control on the CPU.

Load Average

It is the average number of processes residing in the ready queue waiting for their turn to get into the CPU.

Response Time

Amount of time it takes from when a request was submitted until the first response is produced. Remember, it is the time till the first response and not the completion of process execution (final response).

In general CPU utilization and Throughput are maximized and other factors are reduced for proper optimization.

DISPATCHER

Another component involved in the CPU scheduling function is the **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 from where it left last time.

The dispatcher should be as fast as possible, given that it is invoked during every process switch. The time taken by the dispatcher to stop one process and start another process is known as the **Dispatch Latency**. Dispatch Latency can be explained using the below

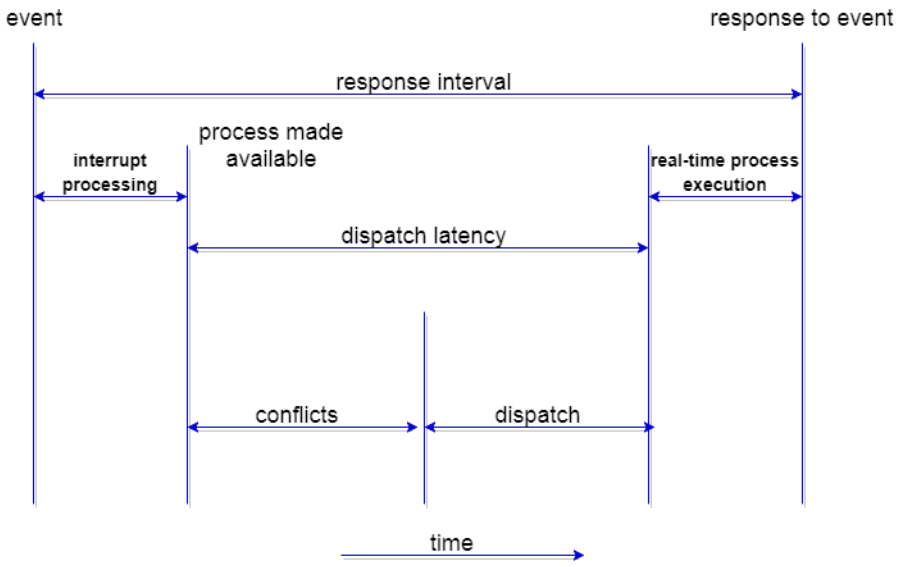


Figure: 2.2 Dispatch Latency

CPU SCHEDULING

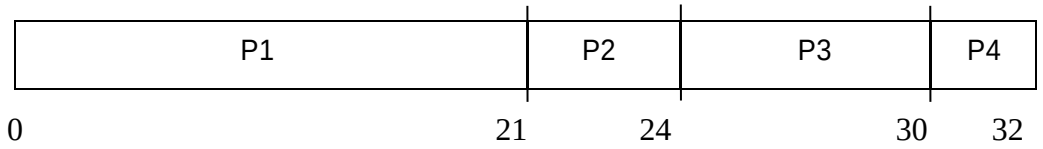
FIRST COME FIRST SERVE

First Come First Serve (FCFS) scheduling algorithm simply schedules the jobs according to their arrival time. The job which comes first in the ready queue will get the CPU first. The lesser the arrival time of the job, the sooner will the job get the CPU. FCFS scheduling may cause the problem of starvation if the burst time of the first process is the longest among all the jobs.

- First Come First Serve is just like FIFO (First in First out) Queue data structure, where the data element which is added to the queue first, is the one who leaves the queue first.
- This is used in Batch Systems.
- It's easy to understand and implement programmatically, using a Queue data structure, where a new process enters through the tail of the queue, and the scheduler selects process from the head of the queue.
- A perfect real life example of FCFS scheduling is buying tickets at ticket counter.

Consider the processes P1, P2, P3, P4 given in the below table, arrives for execution in the same order, with Arrival Time 0, and given Burst Time, let's find the average waiting time using the FCFS scheduling algorithm.

PROCESS	BURST TIME
P1	21
P2	3
P3	6
P4	2



For the above given processes, first P1 will be provided with the CPU resources

- Hence, waiting time for P1 will be 0
- P1 requires 21 ms for completion, hence waiting time for P2 will be 21 ms
- Similarly, waiting time for process P3 will be execution time of P1 + execution time for P2, which will be (21 + 3) ms = 24 ms.
- For process P4 it will be the sum of execution times of P1, P2 and P3.

PROCESS	BURST TIME	WAITING TIIME
P1	21	0
P2	3	21
P3	6	24
P4	2	30

AVERAGE WAITING TIME = (0+21+24+30)/4 =18.75

ADVANTAGES

1. Suitable for batch system
2. FCFS is pretty simple and easy to implement.
3. Eventually, every process will get a chance to run, so starvation doesn't occur.

DISADVANTAGES

1. The scheduling method is non preemptive, the process will run to the completion.

2. Due to the non-preemptive nature of the algorithm, the problem of starvation may occur.
3. Although it is easy to implement, but it is poor in performance since the average waiting time is higher as compare to other scheduling algorithms.

IMLPEMENTATION IN C PROGRAM

```
#include<stdio.h>
int main()
{
    int n,bt[20],wt[20],tat[20],avwt=0,avtat=0,i,j;
    printf("Enter total number of processes(maximum 20):");
    scanf("%d",&n);
     printf("\nEnter Process Burst Time\n");
    for(i=0;i<n;i++)
    {
        printf("P[%d]:",i+1);
        scanf("%d",&bt[i]);
    }
     wt[0]=0;    //waiting time for first process is 0

    //calculating waiting time
    for(i=1;i<n;i++)
    {
        wt[i]=0;
        for(j=0;j<i;j++)
            wt[i]+=bt[j];
    }
printf("\nProcess\t\tBurst Time\tWaiting Time\tTurnaround
Time");
     //calculating turnaround time
    for(i=0;i<n;i++)
    {
        tat[i]=bt[i]+wt[i];
        avwt+=wt[i];
        avtat+=tat[i];
   printf("\nP[%d]\t\t%d\t\t%d\t\t%d",i+1,bt[i],wt[i],tat[i]);
    }
     avwt/=i;
    avtat/=i;
    printf("\n\nAverage Waiting Time:%d",avwt);
    printf("\nAverage Turnaround Time:%d",avtat);
     return 0;}
```

OUTPUT

Enter total number of processes(maximum 20):4
Enter Process Burst Time
P[1]:21
P[2]:3
P[3]:6
P[4]:2

Process	Burst Time	Waiting Time	Turnaround Time
P[1]	21	0	21
P[2]	3	21	24
P[3]	6	24	30
P[4]	2	30	32

Average Waiting Time:18.750000

CONVOY EFFECT

- Convoy Effect is a situation where many processes, who need to use a resource for short time are blocked by one process holding that resource for a long time.
- This essentially leads to poor utilization of resources and hence poor performance.

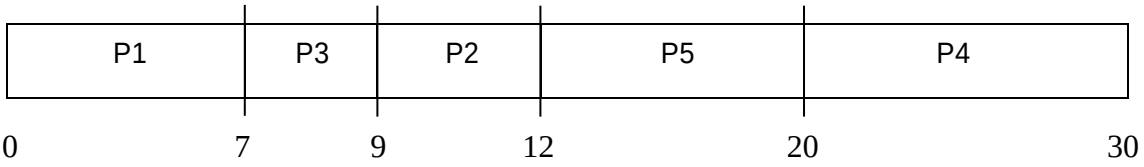
SHORTEST JOB FIRST

A diverse approach to CPU scheduling is the technique of shortest-job-first (SJF) scheduling algorithm which links with each process the length of the process's next CPU burst. If the CPU is available, it is assigned to the process that has the minimum next CPU burst. If the subsequent CPU bursts of two processes become the same, then FCFS scheduling is used to break the tie.

- SJF scheduling algorithm, schedules the processes according to their burst time.
- In SJF scheduling, the process with the lowest burst time, among the list of available processes in the ready queue, is going to be scheduled next.
- However, it is very difficult to predict the burst time needed for a process hence this algorithm is very difficult to implement in the system.

In the following example, there are five jobs named as P1, P2, P3, P4 and P5. Their arrival time and burst time are given in the table below.

PID	Arrival Time	Burst Time
1	0	7
2	3	3
3	4	2
4	7	10
5	9	8

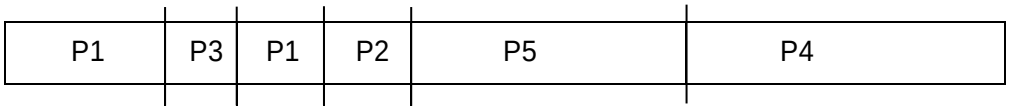


0 7 9 12 20 30

For the above given processes, first P1 will be provided with the CPU resources based on Non preemptive scheduling

- Hence, waiting time for P1 will be 0
- P1 requires 7 ms for completion, CPU looks for the net process based on the lowest burst time. Compare with P2, P3 and P4, P3 has the smallest burst time, so P3 will be executed next.
- Similarly, Next will be P2, since it has the smallest burst time.
- Next will be P5 and at last P4.

Average Waiting Time=7.8



0 4 6 9 12 20 30

For the above given processes, first P1 will be provided with the CPU resources based on preemptive scheduling

- Hence, waiting time for P1 will be 0
- P1 requires 7 ms for completion, but P3 arrives CPU looks for the net process based on the lowest burst time. Compare with P2, P3 and P4, P3 has the smallest burst time, so P3 will be executed next.
- Similarly, Next will be P2, since it has the smallest burst time.

- Next will be P5 and at last P4.

Advantages

- short processes are executed first and then followed by longer processes.
- The throughput is increased because more processes can be executed in less amount of time.

Disadvantages:

- The time taken by a process must be known by the CPU beforehand, which is not possible.
- Longer processes will have more waiting time, eventually they'll suffer starvation.

Sample program

```
#include<stdio.h>
 void main()
{
    int bt[20],p[20],wt[20],tat[20],i,j,n,total=0,pos,temp;
    float avg_wt,avg_tat;
    printf("Enter number of process:");
    scanf("%d",&n);

    printf("\nEnter Burst Time:\n");
    for(i=0;i<n;i++)
    {
        printf("p%d:",i+1);
        scanf("%d",&bt[i]);
        p[i]=i+1;           //contains process number
    }
     //sorting burst time in ascending order using selection
             sort
    for(i=0;i<n;i++)
    {
        pos=i;
        for(j=i+1;j<n;j++)
        {
            if(bt[j]<bt[pos])
                pos=j;
        }

        temp=bt[i];
        bt[i]=bt[pos];
        bt[pos]=temp;

        temp=p[i];
        p[i]=p[pos];
```

```
        p[pos]=temp;
    }
     wt[0]=0;    //waiting time for first process will be zero
     //calculate waiting time
    for(i=1;i<n;i++)
    {
        wt[i]=0;
        for(j=0;j<i;j++)
            wt[i]+=bt[j];

        total+=wt[i];
    }
     avg_wt=(float)total/n;      //average waiting time
    total=0;
     printf("\nProcess\t    Burst Time    \tWaiting
Time\tTurnaround Time");
    for(i=0;i<n;i++)
    {
        tat[i]=bt[i]+wt[i];     //calculate turnaround time
        total+=tat[i];
        printf("\np%d\t\t  %d\t\t    %d\t\t\t%d",p[i],bt[i],wt
[i],tat[i]);
    }
    avg_tat=(float)total/n;     //average turnaround time
    printf("\n\nAverage Waiting Time=%f",avg_wt);
    printf("\nAverage Turnaround Time=%f\n",avg_tat);
}
```

Output

Enter number of process:5

Enter Burst Time:
p1:7
p2:3
p3:2
p4:10
p5:8

Process	Burst Time	Waiting Time	Turnaround Time
p3	2	0	2
p2	3	2	5
p1	7	5	12
p5	8	12	20
p4	10	20	30

Average Waiting Time=7.800000
Average Turnaround Time=13.800000

ROUND-ROBIN

The round-robin (RR) scheduling technique is intended mainly for time-sharing systems. This algorithm is related to FCFS scheduling, but pre-emption is included to toggle among processes. A small unit of time which is termed as a time quantum or time slice has to be defined. A 'time quantum' is usually from 10 to 100 milliseconds. The ready queue gets treated with a circular queue. The CPU scheduler goes about the ready queue, allocating the CPU with each process for the time interval which is at least 1-time quantum.

- A fixed time is allotted to each process, called **quantum**, for execution.
- Once a process is executed for given time period that process is preempted and other process executes for given time period.
- Context switching is used to save states of preempted processes.
- If time quantum is very large than RR scheduling algorithm treat as FCFS and if time quantum is small than RR called processor sharing. Processor sharing show to each process that they have their own processor.
- The central concept is time switching in RR scheduling. If the context switch time is 10 percent of the time quantum then about 10 percent time will be spent in context switching.
- The ready queue is maintained as a circular queue, so when all processes have had a turn, then the scheduler gives the first process another turn, and so on.

Advantages

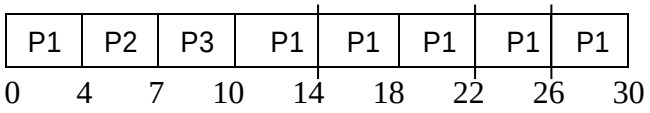
1. It can be actually implementable in the system because it is not depending on the burst time.
2. It doesn't suffer from the problem of starvation or convoy effect.
3. All the jobs get a fare allocation of CPU.

Disadvantages

1. The higher the time quantum, the higher the response time in the system.
2. The lower the time quantum, the higher the context switching overhead in the system.
3. Deciding a perfect time quantum is really a very difficult task in the system.

PROCESS	ARRIVAL TIME	BURST TIME
P1	0	24
P2	1	3
P3	2	3

- The performance of RR is sensitive to the time quantum selected. If the quantum is large enough, then RR reduces to the FCFS algorithm; If it is very small, then each process gets 1/nth of the processor time and share the CPU equally.
- But, a real system invokes overhead for every context switch, and the smaller the time quantum the more context switches. Most modern systems use time quantum between 10 and 100 milliseconds, and context switch times on the order of 10 microseconds, so the overhead is small relative to the time quantum.
- Here we taken Time Quantum =4




```
#include<stdio.h>
int main()
{
   int count,j,n,time,remain,flag=0,time_quantum;
  int wait_time=0,turnaround_time=0,at[10],bt[10],rt[10];
  printf("Enter Total Process:\t ");
  scanf("%d",&n);
  remain=n;
  for(count=0;count<n;count++)
  {
    printf("Enter Arrival Time and Burst Time for Process
Process Number %d :",count+1);
    scanf("%d",&at[count]);
    scanf("%d",&bt[count]);
```

```
    rt[count]=bt[count];
  }
  printf("Enter Time Quantum:\t");
  scanf("%d",&time_quantum);
  printf("\n\nProcess\t|Turnaround Time|Waiting Time\n\n");
  for(time=0,count=0;remain!=0;)
  {
    if(rt[count]<=time_quantum && rt[count]>0)
    {
      time+=rt[count];
      rt[count]=0;
      flag=1;
    }
    else if(rt[count]>0)
    {
      rt[count]-=time_quantum;
      time+=time_quantum;
    }
    if(rt[count]==0 && flag==1)
    {
      remain--;
      printf("P[%d]\t|\t%d\t|\t%d\n",count+1,time-
at[count],time-at[count]-bt[count]);
      wait_time+=time-at[count]-bt[count];
      turnaround_time+=time-at[count];
      flag=0;
    }
    if(count==n-1)
      count=0;
    else if(at[count+1]<=time)
      count++;
```

```
    else
      count=0;
  }
  printf("\nAverage Waiting Time= %f\n",wait_time*1.0/n);
  printf("Avg Turnaround Time = %f",turnaround_time*1.0/n);
    return 0;
}
```

OUTPUT

Enter Total Process: 3

Enter Arrival Time and Burst Time for Process Process Number 1 :1

24

Enter Arrival Time and Burst Time for Process Process Number 2 :2

3

Enter Arrival Time and Burst Time for Process Process Number 3 :3

3

Enter Time Quantum: 4

Process	Turnaround Time	Waiting Time
P[2]	5	2
P[3]	7	4
P[1]	29	5

Average Waiting Time= 3.666667

PRIORITY SCHEDULING

Scheduler consider the priority of processes. The priority assigned to each process and CPU allocated to highest priority process. Equal priority processes scheduled in FCFS order.

Priority can be discussed regarding Lower and higher priority. Numbers denote it. We can use 0 for lower priority as well as more top priority. There is not a hard and fast rule to assign numbers to preferences.

Priority Scheduling suffers from a starvation problem. The starvation problem leads to indefinite blocking of a process due to low priority. Every time higher priority process acquires CPU, and Low priority process is still waiting in the waiting queue. The aging technique gives us a solution to overcome this starvation problem in this technique; we increased the priority of process that was waiting in the system for a long time.

Advantages

- The priority of a process can be selected based on memory requirement, time requirement or user preference. For example, a high end game will have better graphics, that means the process which updates the screen in a game will have higher priority so as to achieve better graphics performance.

Disadvantages:

- A second scheduling algorithm is required to schedule the processes which have same priority.
- In preemptive priority scheduling, a higher priority process can execute ahead of an already executing lower priority process. If lower priority process keeps waiting for higher priority processes, starvation occurs.

Now in this example, we are using low numbers to represent higher priority.

PROCESS	BURST TIME	PRIORITY
P1	10	4
P2	4	1
P3	6	3
P4	5	2

Average waiting time= (0+4+9+15) /4=28/4=7

Average time is 7 milliseconds.

```
#include<stdio.h>

int main()
{
```

```
    int
bt[20],p[20],wt[20],tat[20],pr[20],i,j,n,total=0,pos,temp,avg_
wt,avg_tat;
    printf("Enter Total Number of Process:");
    scanf("%d",&n);

    printf("\nEnter Burst Time and Priority\n");
    for(i=0;i<n;i++)
    {
        printf("\nP[%d]\n",i+1);
        printf("Burst Time:");
        scanf("%d",&bt[i]);
        printf("Priority:");
        scanf("%d",&pr[i]);
        p[i]=i+1;           //contains process number
    }
     //sorting burst time, priority and process number in
ascending order using selection sort
    for(i=0;i<n;i++)
    {
        pos=i;
        for(j=i+1;j<n;j++)
        {
            if(pr[j]<pr[pos])
                pos=j;
        }
        temp=pr[i];
        pr[i]=pr[pos];
        pr[pos]=temp;

        temp=bt[i];
        bt[i]=bt[pos];
        bt[pos]=temp;

        temp=p[i];
        p[i]=p[pos];
        p[pos]=temp;
    }

    wt[0]=0;   //waiting time for first process is zero

    //calculate waiting time
    for(i=1;i<n;i++)
    {
        wt[i]=0;
        for(j=0;j<i;j++)
            wt[i]+=bt[j];
```

```
        total+=wt[i];
    }

    avg_wt=total/n;      //average waiting time
    total=0;

    printf("\nProcess\t    Burst Time    \tWaiting
Time\tTurnaround Time");
    for(i=0;i<n;i++)
    {
        tat[i]=bt[i]+wt[i];     //calculate turnaround time
        total+=tat[i];
        printf("\nP[%d]\t\t  %d\t\t
%d\t\t\t%d",p[i],bt[i],wt[i],tat[i]);
    }

    avg_tat=total/n;     //average turnaround time
    printf("\n\nAverage Waiting Time=%d",avg_wt);
    printf("\nAverage Turnaround Time=%d\n",avg_tat);

     return 0;
}
```

OUTPUT

Enter Total Number of Process:4

Enter Burst Time and Priority

P[1]
Burst Time:10
Priority:4

P[2]
Burst Time:4
Priority:1

P[3]
Burst Time:6
Priority:3

P[4]
Burst Time:5
Priority:2

Process	Burst Time	Waiting Time	Turnaround Time
P[2]	4	0	4
P[4]	5	4	9
P[3]	6	9	15
P[1]	10	15	25

Average Waiting Time=7

Average Turnaround Time=13

MULTILEVEL QUEUE SCHEDULING

This Scheduling algorithm has been created for situations in which processes are easily classified into different groups.

1. System Processes
2. Interactive Processes
3. Interactive Editing Processes
4. Batch Processes
5. Student Processes

For example: A common division is made between foreground(or interactive) processes and background (or batch) processes. These two types of processes have different response-time requirements, and so might have different scheduling needs. In addition, foreground processes may have priority over background processes.

A multi-level queue scheduling algorithm partitions the ready queue into several separate queues. The processes are permanently assigned to one queue, generally based on some property of the process, such as memory size, process priority, or process type. Each queue has its own scheduling algorithm.

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

In addition, there must be scheduling among the queues, which is commonly implemented as fixed-priority preemptive scheduling. **For example:** The foreground queue may have absolute priority over the background queue.

Each queue has absolute priority over lower-priority queues. No process in the batch queue, for example, could run unless the queues for system processes, interactive processes, and interactive editing processes were all empty. If an interactive editing process entered the ready queue while a batch process was running, the batch process will be preempted.

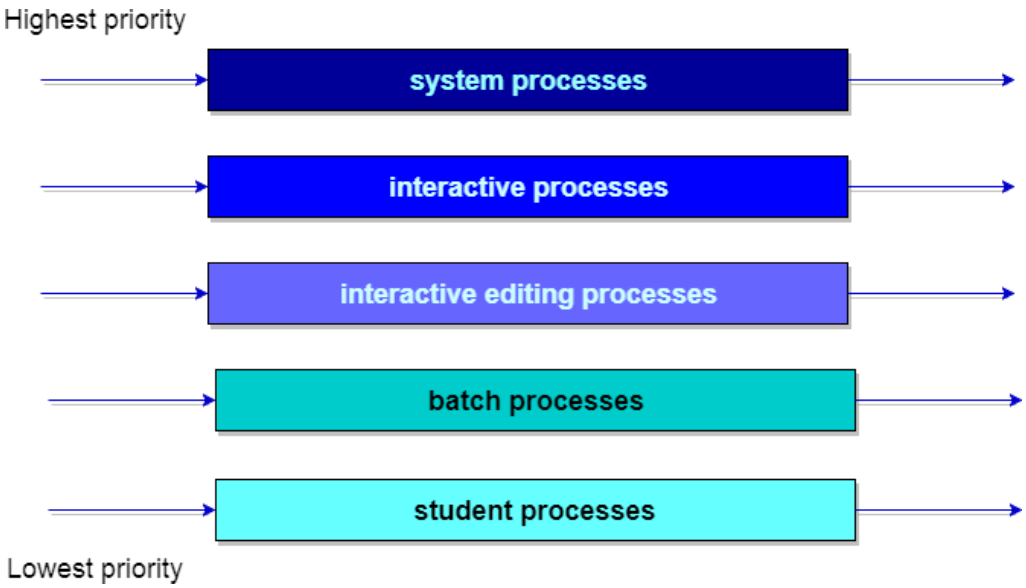


Figure: 2.3 Multi level scheduling

MULTILEVEL FEEDBACK QUEUE SCHEDULING

In a multilevel queue-scheduling algorithm, processes are permanently assigned to a queue on entry to the system. Processes do not move between queues. This setup has the advantage of low scheduling overhead, but the disadvantage of being inflexible.

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. 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.

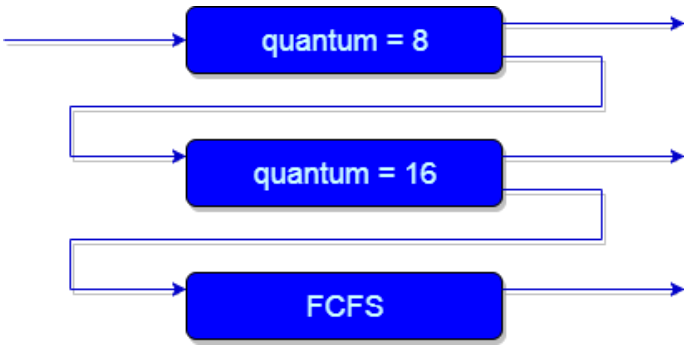


Figure: 2.4 Multi level queue scheduling

An example of a multilevel feedback queue can be seen in the below figure.

In general, 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.

The definition of a multilevel feedback queue scheduler makes it the most general CPU-scheduling algorithm. It can be configured to match a specific system under design. Unfortunately, it also requires some means of selecting values for all the parameters to define the best scheduler. Although a multilevel feedback queue is the **most general scheme**, it is also the **most complex**.

DEADLOCK

INTRODUCTION

In a multiprogramming system, numerous processes get competed for a finite number of resources. Any process requests resources, and as the resources aren't available at that time, the process goes into a waiting state. At times, a waiting process is not at all able again to change its state as other waiting processes detain the resources it has requested. That condition is termed as deadlock. Every process needs some resources to complete its execution. However, the resource is granted in a sequential order.

1. The process requests for some resource.
2. OS grant the resource if it is available otherwise let the process waits.
3. The process uses it and release on the completion.

A Deadlock is a situation where each of the computer process waits for a resource which is being assigned to some another process. In this situation, none of the process gets executed since the resource it needs, is held by some other process which is also waiting for some other resource to be released.

Let us assume that there are three processes P1, P2 and P3. There are three different resources R1, R2 and R3. R1 is assigned to P1, R2 is assigned to P2 and R3 is assigned to P3.

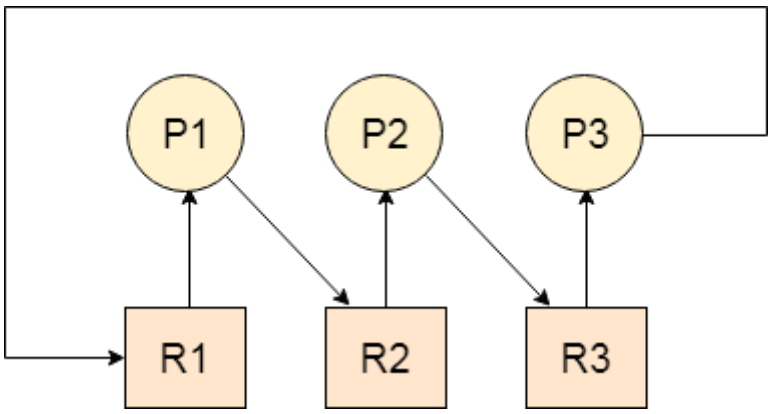


Figure: 2.5 Deadlock Example

Definition: A deadlock happens in operating system when two or more processes need some resource to complete their execution that is held by the other process.

A system model or structure consists of a fixed number of resources to be circulated among some opposing processes. The resources are then partitioned into numerous types, each consisting of some specific quantity of identical instances. Memory space, CPU cycles, directories and files, I/O devices like keyboards, printers and CD-DVD drives are prime examples of resource types. When a system has 2 CPUs, then the resource type CPU got two instances.

Under the standard mode of operation, any process may use a resource in only the below-mentioned sequence:

1. Request: When the request can't be approved immediately (where the case may be when another process is utilizing the resource), then the requesting job must remain waited until it can obtain the resource.
2. Use: The process can run on the resource (like when the resource is a printer, its job/process is to print on the printer).
3. Release: The process releases the resource (like, terminating or exiting any specific process).

2.6.1 REAL TIME EXAMPLES OF DEADLOCK

If a process is given the task of waiting for an event to occur, and if the system includes no provision for signaling that event, then we have a one process deadlock. Several common examples of deadlock are

A Traffic Deadlock:

A number of automobiles are attempting to drive through a busy section of the city, but the traffic has become completely jammed. Traffic comes to a halt, and it is necessary for the police to unwind the jam by slowly and carefully backing cars out of the area. Eventually the traffic begins to flow normally, but not without much annoyance, effort and the loss of considerable time.

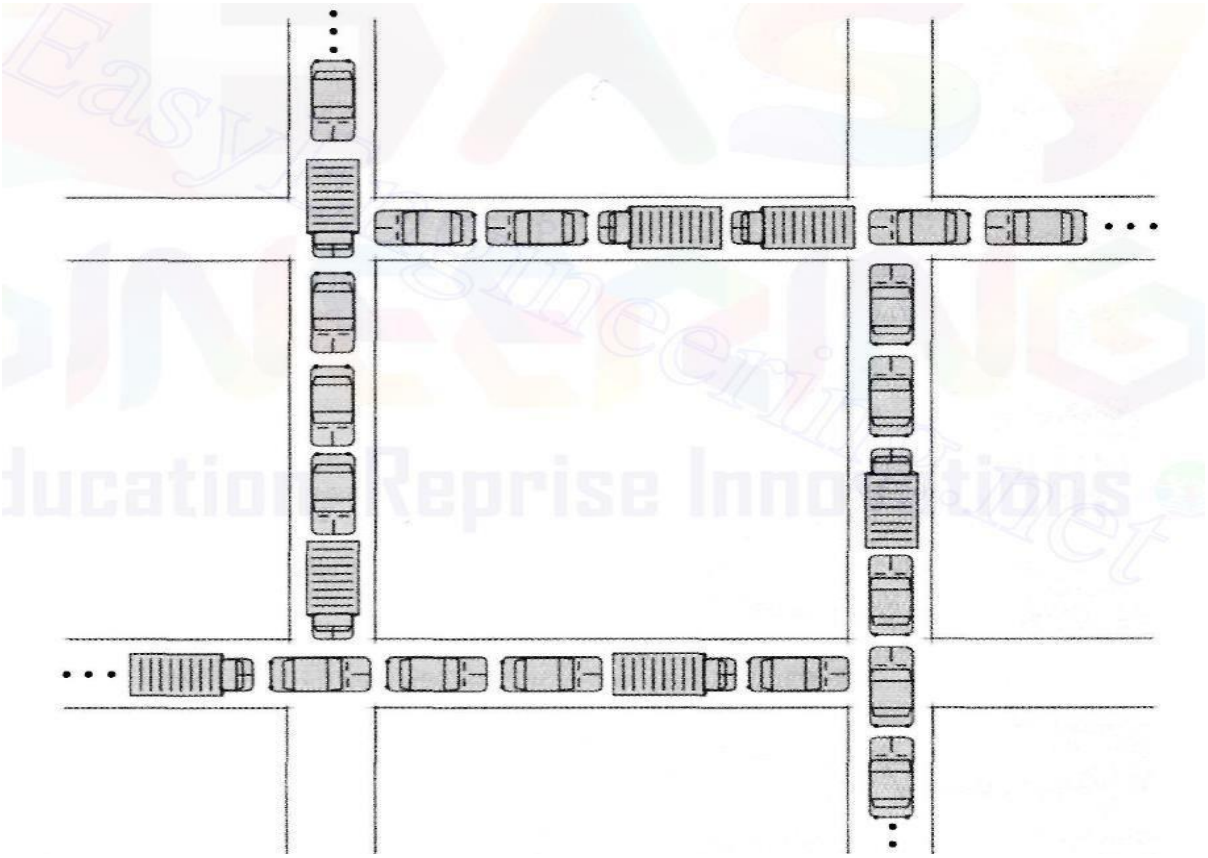


Figure: 2.6 Traffic Jam

B. A simple resource deadlock:

A simple examples of a resource deadlock is illustrated

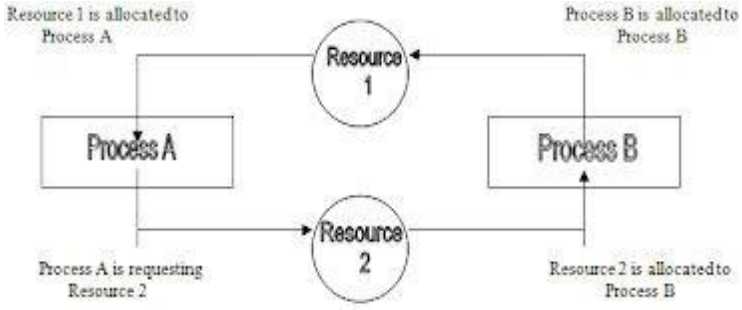


Figure: 2.7 Resource Deadlock

This resource allocation graph shows two processes as rectangles and two resources as circles. An arrow from a resource to a process indicates that the resource belongs to, or has been allocated to the process. An arrow from a process to a resource indicates that the process is requesting, but has not yet been allocated, the resource. The diagram illustrates a

deadlocked system: process A holds Resource 1 and needs Resource 2 to continue. Process B holds Resource 2 and needs Resource 1 to continue. Each process is waiting for the other to free a resource that it will not free until the other frees its resources that it will not do until the other from its resources , etc. This circular wait is characterized of deadlock systems.

C. Deadlock in spooling systems:

Spooling systems are often prone to deadlock. A spooling system is used to improve system throughput by disassociating a program from the slow operating speeds of devices such as printers. For example, if a program sending lines to the printer must wait for each line to be printed before it can transmit the next line, then the program will execute slowly. To speed the program's executing, output lines are routed to a much faster device such as a disk drive where they are temporarily stored until they may be printed. In some spooling systems, the complete output from a program must be available before actual printing can begin. Thus several partially complete jobs generating print lines to a spool file could become deadlocked if the available space fills before any job completes. Unwinding or recovering from such a deadlock might involve restarting the system with a loss of all work performed so far.

SIMPLE REVIEW

1. **Assuming that there are no cars beyond the ellipses in Fig. 2.3, what minimum number of cars would have to back up to relieve the deadlock and which car(s) would they be?**

   In Fig. 2.3, only two cars would need to back up to allow every other car to eventually Move any one of the cars abutting an ellipsis, then the car ahead of that one in the intersection.

2. **If cars could be removed by airlifting in Fig. 2.3, what minimum number of cars, and which one(s), would have to be removed to relieve the deadlock?**

   Only one car has to be removed—namely, any one of the four cars in the intersections.

DEADLOCK CHARACTERIZATION

NECESSARY CONDITIONS FOR DEADLOCKS

A deadlock occurs if the four conditions hold true. But these conditions are not mutually exclusive.

The Coffman conditions are given as follows:

- **Mutual Exclusion**

  There should be a resource that can only be held by one process at a time. In the diagram below, there is a single instance of Resource 1 and it is held by Process 1 only.

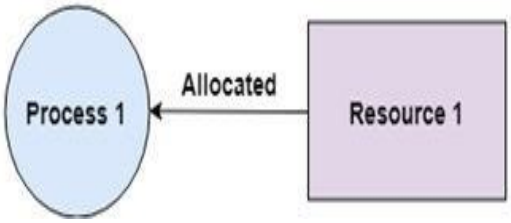


Figure: 2.8 Mutual Exclusion

- **Hold and Wait**

  A process can hold multiple resources and still request more resources from other processes which are holding them. In the diagram given below, Process 2 holds Resource 2 and Resource 3 and is requesting the Resource 1 which is held by Process 1.

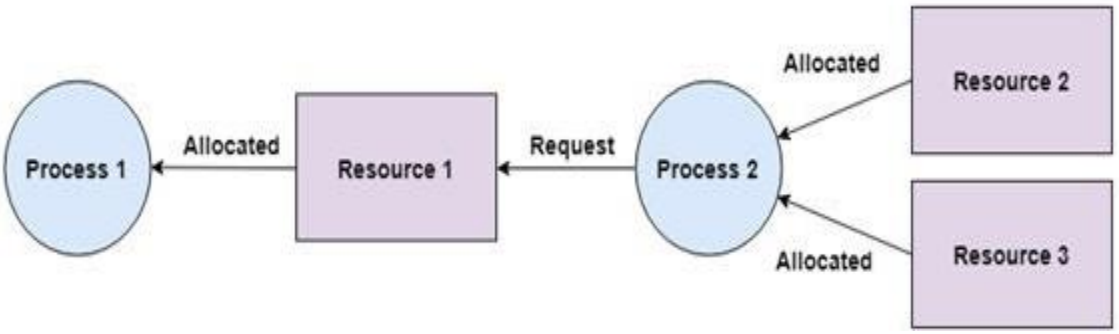


Figure: 2.9 Hold and Wait

- **No Preemption**

  A resource cannot be preempted from a process by force. A process can only release a resource voluntarily. In the diagram below, Process 2 cannot preempt Resource 1 from Process 1. It will only be released when Process 1 relinquishes it voluntarily after its execution is complete.

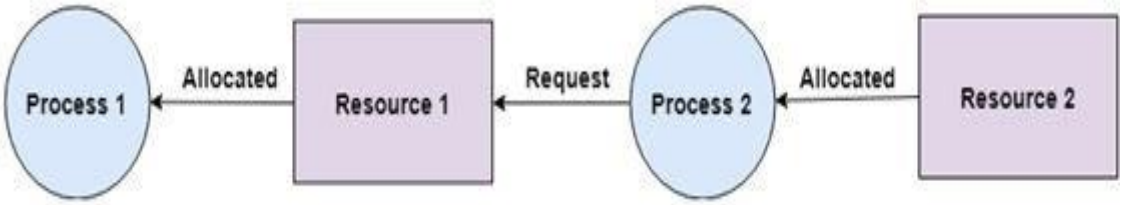


Figure: 2.10 No Preemption

- **Circular Wait**

  A process is waiting for the resource held by the second process, which is waiting for the resource held by the third process and so on, till the last process is waiting for a resource held by the first process. This forms a circular chain. For example: Process 1 is allocated Resource2 and it is requesting Resource 1. Similarly, Process 2 is allocated Resource 1 and it is requesting Resource 2. This forms a circular wait loop.

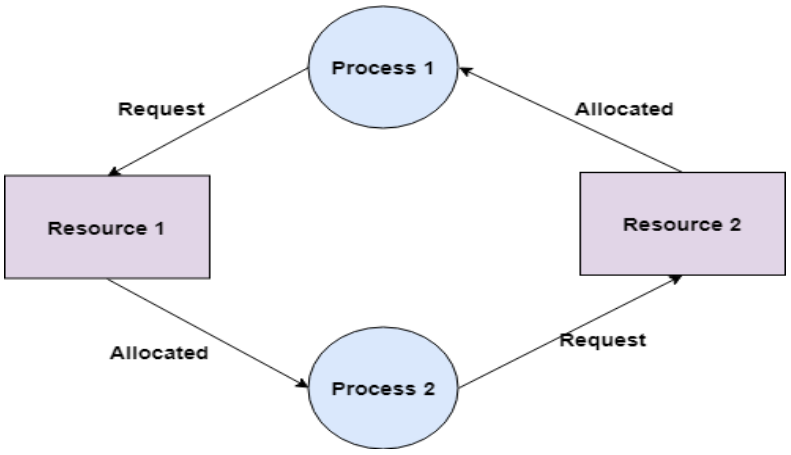


Figure: 2.11 Circular Wait

RESOURCE – ALLOCATION GRAPH

Definition: A resource allocation graph tracks which resource is held by which process and which process is waiting for a resource of a particular type. It is very powerful and simple tool to illustrate how interacting processes can deadlock.

The resource allocation graph is the pictorial representation of the state of a system. As its name suggests, the resource allocation graph is the complete information about all the processes which are holding some resources or waiting for some resources.

It also contains the information about all the instances of all the resources whether they are available or being used by the processes.

In Resource allocation graph, the process is represented by a Circle while the Resource is represented by a rectangle.

In RAG vertices are two types –

1. **Process vertex** – Every process will be represented as a process vertex. Generally, the process will be represented with a circle.

2. **Resource vertex** – Every resource will be represented as a resource vertex. It is also two types –

- Single instance type resource – It represents as a box, inside the box, there will be one dot. So the number of dots indicates how many instances are present of each resource type.
- Multi-resource instance type resource – It also represents as a box, inside the box, there will be many dots present.

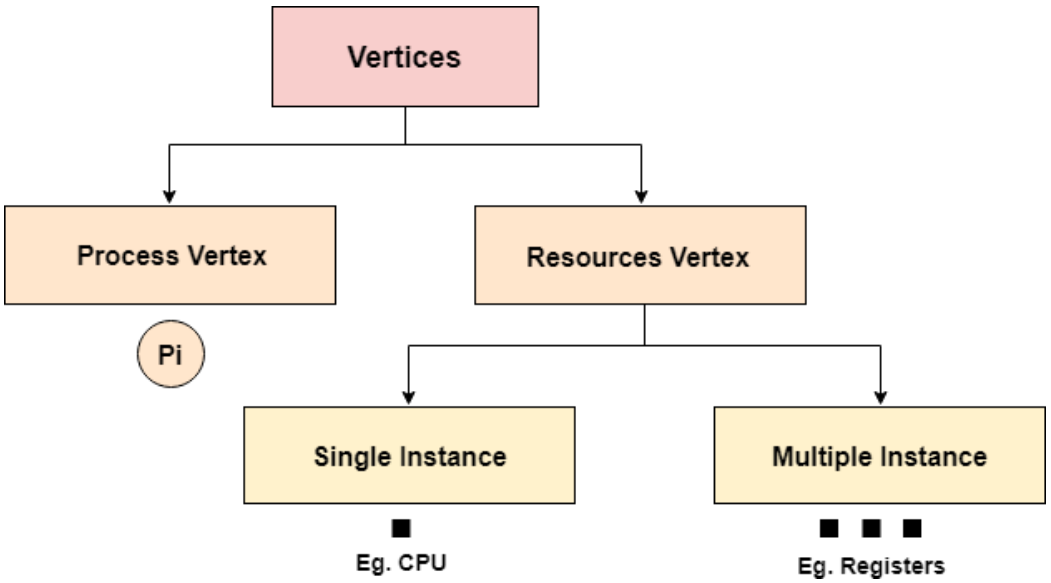


Figure: 2.12 Process and Resources

Vertices are mainly of two types, Resource and process. Each of them will be represented by a different shape. Circle represents process while rectangle represents resource.

A resource can have more than one instance. Each instance will be represented by a dot inside the rectangle.

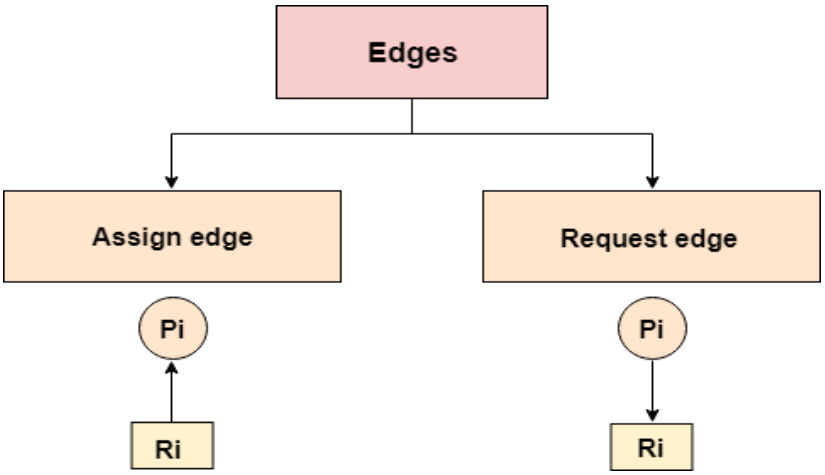


Figure: 2.13 Edges

Edges in RAG are also of two types, one represents assignment and other represents the wait of a process for a resource. The above image shows each of them.

A resource is shown as assigned to a process if the tail of the arrow is attached to an instance to the resource and the head is attached to a process.

A process is shown as waiting for a resource if the tail of an arrow is attached to the process while the head is pointing towards the resource.

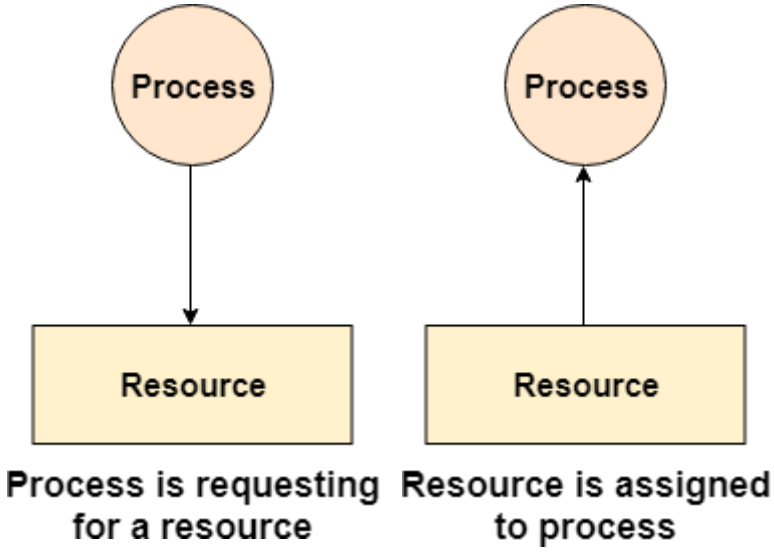


Figure: 2.14 Resource to process Allocation

Example

Let's consider 3 processes P1, P2 and P3, and two types of resources R1 and R2. The resources are having 1 instance each.

According to the graph, R1 is being used by P1, P2 is holding R2 and waiting for R1, P3 is waiting for R1 as well as R2.

The graph is deadlock free since no cycle is being formed in the graph.

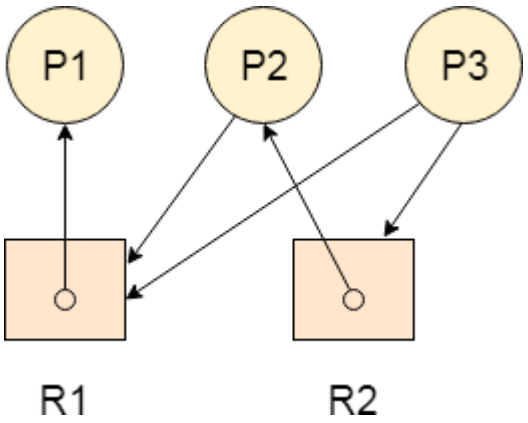


Figure: 2.15 Resource Allocation Graph

If a cycle is being formed in a Resource allocation graph where all the resources have the single instance then the system is deadlocked.

In Case of Resource allocation graph with multi-instanced resource types, Cycle is a necessary condition of deadlock but not the sufficient condition.

The following example contains three processes P1, P2, P3 and three resources R2, R2, R3. All the resources are having single instances each.

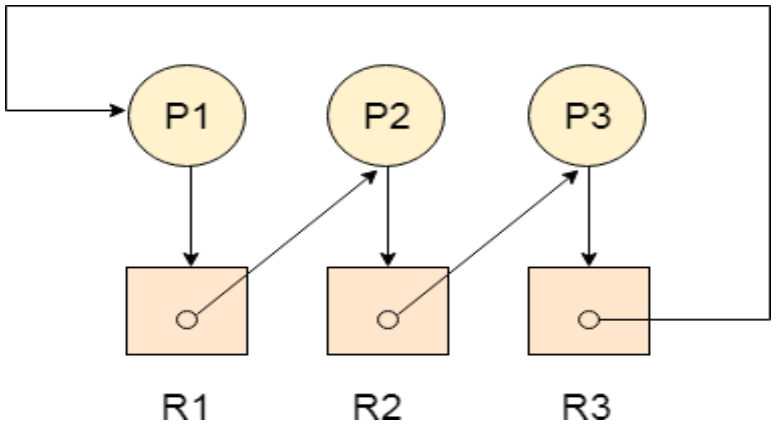


Figure: 2.16 Resource Allocation Graph with Deadlock

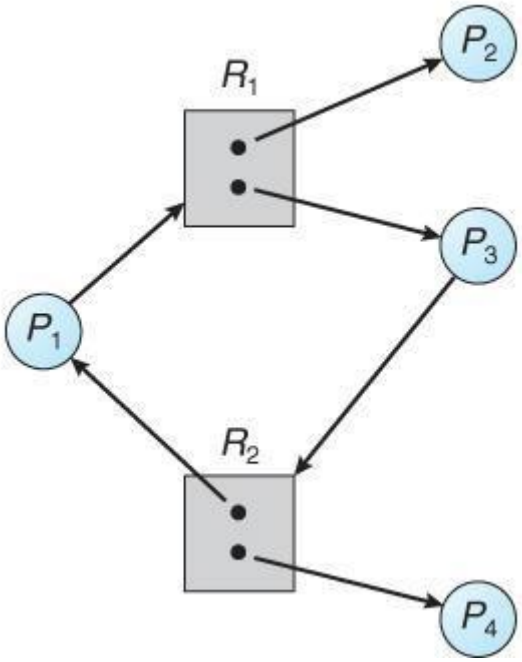


Figure: 2.17 Resource Allocation Graph with a cycle but no Deadlock

METHODS FOR HANDLING DEADLOCK

Generally speaking there are three ways of handling deadlocks:

- Deadlock prevention or avoidance - Do not allow the system to get into a deadlocked state.
- Deadlock detection and recovery - Abort a process or preempt some resources when deadlocks are detected.
- Ignore the problem all together - If deadlocks only occur once a year or so, it may be better to simply let them happen and reboot as necessary than to incur the constant overhead and system performance penalties associated with deadlock prevention or detection. This is the approach that both Windows and UNIX take.

In order to avoid deadlocks, the system must have additional information about all processes. In particular, the system must know what resources a process will or may request in the future. (Ranging from a simple worst-case maximum to a complete resource request and release plan for each process, depending on the particular algorithm.)

Deadlock detection is fairly straightforward, but deadlock recovery requires either aborting processes or preempting resources, neither of which is an attractive alternative.

If deadlocks are neither prevented nor detected, then when a deadlock occurs the system will gradually slow down, as more and more processes become stuck waiting for resources

currently held by the deadlock and by other waiting processes. Unfortunately this slowdown can be indistinguishable from a general system slowdown when a real-time process has heavy computing needs.

DEADLOCK PREVENTION

Deadlocks can be prevented by preventing at least one of the four required conditions:

Mutual Exclusion

- Shared resources such as read-only files do not lead to deadlocks.
- Unfortunately some resources, such as printers and tape drives, require exclusive access by a single process.

Hold and Wait

- To prevent this condition processes must be prevented from holding one or more resources while simultaneously waiting for one or more others. There are several possibilities for this:
    - Require that all processes request all resources at one time. This can be wasteful of system resources if a process needs one resource early in its execution and doesn't need some other resource until much later.
    - Require that processes holding resources must release them before requesting new resources, and then re-acquire the released resources along with the new ones in a single new request. This can be a problem if a process has partially completed an operation using a resource and then fails to get it re-allocated after releasing it.
    - Either of the methods described above can lead to starvation if a process requires one or more popular resources.

No Preemption

- Preemption of process resource allocations can prevent this condition of deadlocks, when it is possible.
    - One approach is that if a process is forced to wait when requesting a new resource, then all other resources previously held by this process are implicitly released, (preempted), forcing this process to re-acquire the old resources along with the new resources in a single request, similar to the previous discussion.

- Another approach is that when a resource is requested and not available, then the system looks to see what other processes currently have those resources *and* are themselves blocked waiting for some other resource. If such a process is found, then some of their resources may get preempted and added to the list of resources for which the process is waiting.
- Either of these approaches may be applicable for resources whose states are easily saved and restored, such as registers and memory, but are generally not applicable to other devices such as printers and tape drives.

Circular Wait

- One way to avoid circular wait is to number all resources, and to require that processes request resources only in strictly increasing (or decreasing) order.
- In other words, in order to request resource Rj, a process must first release all Ri such that i >= j.
- One big challenge in this scheme is determining the relative ordering of the different resources

DEADLOCK AVOIDANCE

- The general idea behind deadlock avoidance is to prevent deadlocks from ever happening, by preventing at least one of the aforementioned conditions.
- This requires more information about each process, AND tends to lead to low device utilization. (i.e. it is a conservative approach.)
- In some algorithms the scheduler only needs to know the *maximum* number of each resource that a process might potentially use. In more complex algorithms the scheduler can also take advantage of the *schedule* of exactly what resources may be needed in what order.
- When a scheduler sees that starting a process or granting resource requests may lead to future deadlocks, then that process is just not started or the request is not granted.
- A resource allocation ***state*** is defined by the number of available and allocated resources and the maximum requirements of all processes in the system.

Safe State

- A state is ***safe*** if the system can allocate all resources requested by all processes (up to their stated maximums) without entering a deadlock state.

- More formally, a state is safe if there exists a ***safe sequence*** of processes {P0, P1, P2, ..., Pn } such that all of the resource requests for Pi can be granted using the resources currently allocated to Pi and all processes Pj where j < i. (i.e. if all the processes prior to Pi finish and free up their resources, then Pi will be able to finish also, using the resources that they have freed up.)
- If a safe sequence does not exist, then the system is in an unsafe state, which ***MAY*** lead to deadlock. (All safe states are deadlock free, but not all unsafe states lead to deadlocks.)

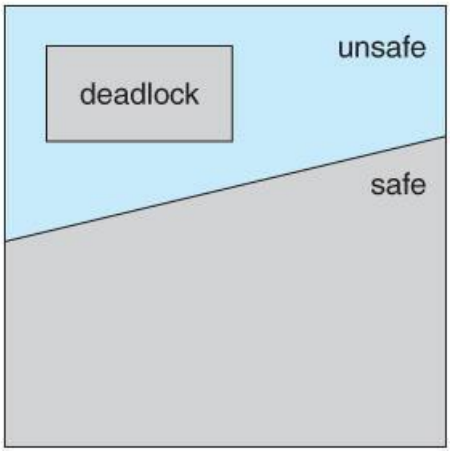


Figure 2.18 - Safe, unsafe, and deadlocked state spaces.

- For example, consider a system with 12 tape drives, allocated as follows. Is this a safe state? What is the safe sequence?

	Maximum Needs	**Current Allocation**
P0	10	5
P1	4	2
P2	9	2

- What happens to the above table if process P2 requests and is granted one more tape drive?
- Key to the safe state approach is that when a request is made for resources, the request is granted only if the resulting allocation state is a safe one.

Resource-Allocation Graph Algorithm

- If resource categories have only single instances of their resources, then deadlock states can be detected by cycles in the resource-allocation graphs.
- In this case, unsafe states can be recognized and avoided by augmenting the resource-allocation graph with ***claim edges***, noted by dashed lines, which point from a process to a resource that it may request in the future.
- In order for this technique to work, all claim edges must be added to the graph for any particular process before that process is allowed to request any resources. (Alternatively, processes may only make requests for resources for which they have already established claim edges, and claim edges cannot be added to any process that is currently holding resources.)
- When a process makes a request, the claim edge Pi->Rj is converted to a request edge. Similarly when a resource is released, the assignment reverts back to a claim edge.
- This approach works by denying requests that would produce cycles in the resource-allocation graph, taking claim edges into effect.
- Consider for example what happens when process P2 requests resource R2:

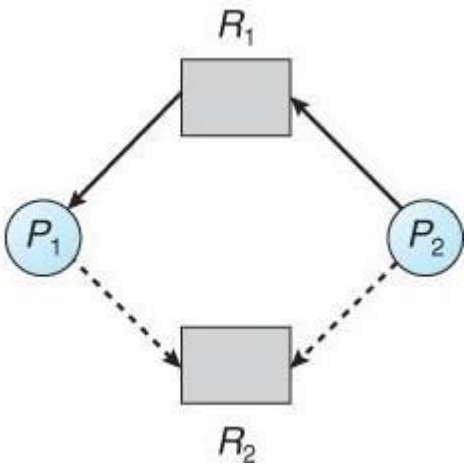


Figure 2.19 - Resource allocation graph for deadlock avoidance

- The resulting resource-allocation graph would have a cycle in it, and so the request cannot be granted.

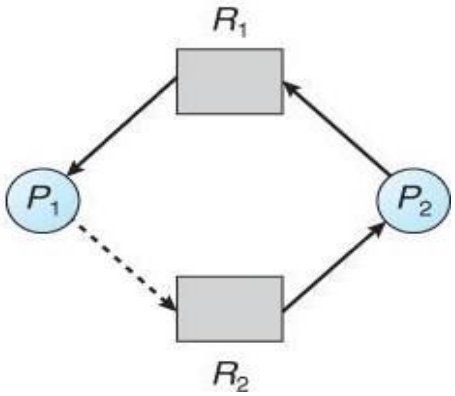


Figure 2.20 - An unsafe state in a resource allocation graph

Banker's Algorithm

- For resource categories that contain more than one instance the resource-allocation graph method does not work, and more complex (and less efficient) methods must be chosen.
- The Banker's Algorithm gets its name because it is a method that bankers could use to assure that when they lend out resources they will still be able to satisfy all their clients. (A banker won't loan out a little money to start building a house unless they are assured that they will later be able to loan out the rest of the money to finish the house.)
- When a process starts up, it must state in advance the maximum allocation of resources it may request, up to the amount available on the system.
- When a request is made, the scheduler determines whether granting the request would leave the system in a safe state. If not, then the process must wait until the request can be granted safely.
- The banker's algorithm relies on several key data structures: (where n is the number of processes and m is the number of resource categories.)
    - Available[m] indicates how many resources are currently available of each type.
    - Max[n][m] indicates the maximum demand of each process of each resource.
    - Allocation[n][m] indicates the number of each resource category allocated to each process.
    - Need[n][m] indicates the remaining resources needed of each type for each process. (Note that Need[i][j] = Max[i][j] - Allocation[i][j] for all i, j.)

- For simplification of discussions, we make the following notations / observations:
    - One row of the Need vector, Need[i], can be treated as a vector corresponding to the needs of process i, and similarly for Allocation and Max.
    - A vector X is considered to be <= a vector Y if X[i] <= Y[i] for all i.

Safety Algorithm

- In order to apply the Banker's algorithm, we first need an algorithm for determining whether or not a particular state is safe.
- This algorithm determines if the current state of a system is safe, according to the following steps:
    1. Let Work and Finish be vectors of length m and n respectively.
        - Work is a working copy of the available resources, which will be modified during the analysis.
        - Finish is a vector of booleans indicating whether a particular process can finish. (or has finished so far in the analysis.)
        - Initialize Work to Available, and Finish to false for all elements.
    2. Find an i such that both (A) Finish[i] == false, and (B) Need[i] < Work. This process has not finished, but could with the given available working set. If no such i exists, go to step 4.
    3. Set Work = Work + Allocation[i], and set Finish[i] to true. This corresponds to process i finishing up and releasing its resources back into the work pool. Then loop back to step 2.
    4. If finish[i] == true for all i, then the state is a safe state, because a safe sequence has been found.
- (JTB's Modification:
    1. In step 1. instead of making Finish an array of booleans initialized to false, make it an array of ints initialized to 0. Also initialize an int s = 0 as a step counter.
    2. In step 2, look for Finish[i] == 0.
    3. In step 3, set Finish[i] to ++s. S is counting the number of finished processes.
    4. For step 4, the test can be either Finish[i] > 0 for all i, or s >= n. The benefit of this method is that if a safe state exists, then Finish[ ] indicates one safe sequence (of possibly many.))

Resource-Request Algorithm (The Bankers Algorithm)

- Now that we have a tool for determining if a particular state is safe or not, we are now ready to look at the Banker's algorithm itself.
- This algorithm determines if a new request is safe, and grants it only if it is safe to do so.
- When a request is made (that does not exceed currently available resources), pretend it has been granted, and then see if the resulting state is a safe one. If so, grant the request, and if not, deny the request, as follows:
    1. Let Request[n][m] indicate the number of resources of each type currently requested by processes. If Request[i] > Need[i] for any process i, raise an error condition.
    2. If Request[i] > Available for any process i, then that process must wait for resources to become available. Otherwise the process can continue to step 3.
    3. Check to see if the request can be granted safely, by pretending it has been granted and then seeing if the resulting state is safe. If so, grant the request, and if not, then the process must wait until its request can be granted safely. The procedure for granting a request (or pretending to for testing purposes) is:
        - Available = Available - Request
        - Allocation = Allocation + Request
        - Need = Need - Request

An Illustrative Example

Consider a system with 5 processes (P_0 ... P_4) and 3 resources types (A(10) B(5) C(7))

Resource-allocation state at time t_0:

Process	**Allocation**			**Max**			**Need**			**Available**		
	A	B	C	A	B	C	A	B	C	A	B	C
P_0	0	1	0	7	5	3	7	4	3	3	3	2
P_1	2	0	0	3	2	2	1	2	2			
P_2	3	0	2	9	0	2	6	0	0			
P_3	2	1	1	2	2	2	0	1	1			
P_4	0	0	2	4	3	3	4	3	1			

Is the system in a safe state? If so, which sequence satisfies the safety criteria?

< P_1, P_3, P_4, P_2, P_0 >

Now suppose, P_1 requests an additional instance of A and 2 more instances of type C.

request[1] = (1,0,2)

1. check if request[1] <= need[i] (yes)
2. check if request[1] <= available[i] (yes)
3. do pretend updates to the state

Process	Allocation			Max			Need			Available		
	A	B	C	A	B	C	A	B	C	A	B	C
P_0	0	1	0	7	5	3	7	4	3	3	3	2
P_1	3	0	2	3	2	2	0	2	0			
P_2	3	0	2	9	0	2	6	0	0			
P_3	2	1	1	2	2	2	0	1	1			
P_4	0	0	2	4	3	3	4	3	1			

Is the system in a safe state? If so, which sequence satisfies the safety criteria?

<P_1, P_3, P_4, P_0, P_2>

Hence, we immediately grant the request.

Do it Yourself

Will a request of (3,3,0) by P4 be granted?

Will a request of (0,2,0) by P0 be granted?

DEADLOCK DETECTION

- **Definition: Deadlock detection** is the process of determining that a deadlock exists
- and identifying the processes and resources involved in the deadlock.
-

- If deadlocks are not avoided, then another approach is to detect when they have occurred and recover somehow.
- In addition to the performance hit of constantly checking for deadlocks, a policy / algorithm must be in place for recovering from deadlocks, and there is potential for lost work when processes must be aborted or have their resources preempted.

Single Instance of Each Resource Type

- If each resource category has a single instance, then we can use a variation of the resource-allocation graph known as a ***wait-for graph***.

- A wait-for graph can be constructed from a resource-allocation graph by eliminating the resources and collapsing the associated edges, as shown in the figure below.
- An arc from Pi to Pj in a wait-for graph indicates that process Pi is waiting for a resource that process Pj is currently holding.

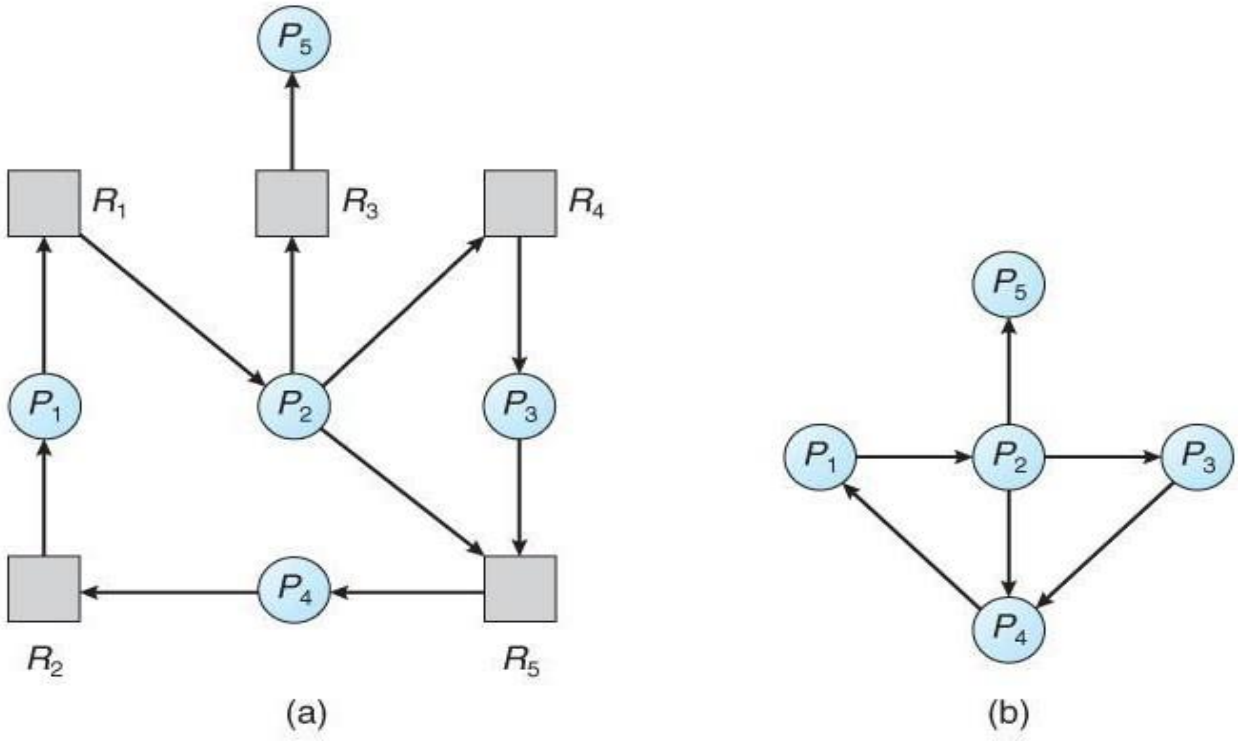


Figure 2.21 - (a) Resource allocation graph. (b) Corresponding wait-for graph

- As before, cycles in the wait-for graph indicate deadlocks.
- This algorithm must maintain the wait-for graph, and periodically search it for cycles.

Several Instances of a Resource Type

- The detection algorithm outlined here is essentially the same as the Banker's algorithm, with two subtle differences:
    - In step 1, the Banker's Algorithm sets Finish[i] to false for all i. The algorithm presented here sets Finish[i] to false only if Allocation[i] is not zero. If the currently allocated resources for this process are zero, the algorithm sets Finish[i] to true. This is essentially assuming that IF all of the other processes can finish, then this process can finish also. Furthermore, this algorithm is specifically looking for which processes are involved in a deadlock situation, and a process that does not have any resources allocated

cannot be involved in a deadlock, and so can be removed from any further consideration.

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

- (Note: An alternative method was presented above, in which Finish held integers instead of booleans. This vector would be initialized to all zeros, and then filled with increasing integers as processes are detected which can finish. If any processes are left at zero when the algorithm completes, then there is a deadlock, and if not, then the integers in finish describe a safe sequence. To modify this algorithm to match this section of the text, processes with allocation = zero could be filled in with N, N - 1, N - 2, etc. in step 1, and any processes left with Finish = 0 in step 4 are the deadlocked processes.)
- Consider, for example, the following state, and determine if it is currently deadlocked:

	Allocation	*Request*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

- Now suppose that process P2 makes a request for an additional instance of type C, yielding the state shown below. Is the system now deadlocked?

	Allocation	Request	Available
	A B C	A B C	A B C
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 1	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

Detection-Algorithm Usage

- When should the deadlock detection be done? Frequently, or infrequently?
- The answer may depend on how frequently deadlocks are expected to occur, as well as the possible consequences of not catching them immediately. (If deadlocks are not removed immediately when they occur, then more and more processes can "back up" behind the deadlock, making the eventual task of unblocking the system more difficult and possibly damaging to more processes.)
- There are two obvious approaches, each with trade-offs:
    1. Do deadlock detection after every resource allocation which cannot be immediately granted. This has the advantage of detecting the deadlock right away, while the minimum numbers of processes are involved in the deadlock. (One might consider that the process whose request triggered the deadlock condition is the "cause" of the deadlock, but realistically all of the processes in the cycle are equally responsible for the resulting deadlock.) The down side of this approach is the extensive overhead and performance hit caused by checking for deadlocks so frequently.
    2. Do deadlock detection only when there is some clue that a deadlock may have occurred, such as when CPU utilization reduces to 40% or some other magic number. The advantage is that deadlock detection is done much less frequently, but the down side is that it becomes impossible to detect the processes involved in the original deadlock, and so deadlock recovery can be more complicated and damaging to more processes.
    3. (As I write this, a third alternative comes to mind: Keep a historical log of resource allocations, since that last known time of no deadlocks. Do deadlock

checks periodically (Once an hour or when CPU usage is low?), and then use the historical log to trace through and determine when the deadlock occurred and what processes caused the initial deadlock. Unfortunately I'm not certain that breaking the original deadlock would then free up the resulting log jam.)

Weaknesses in the Banker's Algorithm

The Banker's Algorithm is compelling because it allows processes to proceed that might have had to wait under a deadlock prevention situation. But the algorithm has a number of weaknesses.

- It requires that there be a fixed number of resources to allocate. Because resources frequently require service, due to breakdowns or preventive maintenance, we cannot count on the number of resources remaining fixed. Similarly, operating systems that support hot swappable devices (e.g., USB devices) allow the number of resources to vary dynamically.
- The algorithm requires that the population of processes remains fixed. This, too, is unreasonable. In today's interactive and multiprogrammed systems, the process population is constantly changing.
- The algorithm requires that the banker (i.e., the system) grant all requests within a "finite time." Clearly, much better guarantees than this are needed in real systems, especially real-time systems.
- Similarly, the algorithm requires that clients (i.e., processes) repay all loans (i.e., return all resources) within a "finite time." Again, much better guarantees than this are needed in real systems.
- The algorithm requires that processes state their maximum needs in advance. With resource allocation becoming increasingly dynamic, it is becoming more difficult to know a process's maximum needs. Indeed, one main benefit of today's high-level programming languages and "friendly" graphical user interfaces is that users are not required to know such low level details as resource use. The user or programmer expects the system to "print the file" or "send the message" and should not need to worry about what resources the system might need to employ to honor such requests.

For the reasons stated above, Banker's Algorithm is not implemented in today's operating systems. In fact, few systems can afford the overhead incurred by deadlock avoidance strategies.

RECOVERY FROM DEADLOCK

There are three basic approaches to recovery from deadlock:

1. Inform the system operator, and allow him/her to take manual intervention.
2. Terminate one or more processes involved in the deadlock
3. Preempt resources.

Process Termination

Two basic approaches, both of which recover resources allocated to terminated processes:

- Terminate all processes involved in the deadlock. This definitely solves the deadlock, but at the expense of terminating more processes than would be absolutely necessary.
- Terminate processes one by one until the deadlock is broken. This is more conservative, but requires doing deadlock detection after each step.

In the latter case there are many factors that can go into deciding which processes to terminate next:

1. Process priorities.
2. How long the process has been running, and how close it is to finishing.
3. How many and what type of resources is the process holding. (Are they easy to preempt and restore?)
4. How many more resources does the process need to complete.
5. How many processes will need to be terminated
6. Whether the process is interactive or batch.
7. (Whether or not the process has made non-restorable changes to any resource.)

Resource Preemption

- When preempting resources to relieve deadlock, there are three important issues to be addressed:
    1. **Selecting a victim** - Deciding which resources to preempt from which processes involves many of the same decision criteria outlined above.
    2. **Rollback** - Ideally one would like to roll back a preempted process to a safe state prior to the point at which that resource was originally allocated to the process. Unfortunately it can be difficult or impossible to determine what such

a safe state is, and so the only safe rollback is to roll back all the way back to the beginning. (i.e. abort the process and make it start over.)

3. **Starvation** - How do you guarantee that a process won't starve because its resources are constantly being preempted? One option would be to use a priority system, and increase the priority of a process every time its resources get preempted. Eventually it should get a high enough priority that it won't get preempted any more.

TWO MARKS QUESTIONS AND ANSWERS

1. Define deadlock.

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.

2. What is the sequence in which resources may be utilized?

Under normal mode of operation a process may utilize a resource in the following sequence:

- Request: If the request cannot be granted immediately, then the requesting process must wait until it can acquire the resource.
- Use: The process can operate on the resource.
- Release: The process releases the resource.

3. What are conditions under which a deadlock situation may arise?

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

- Mutual exclusion
- Hold and wait
- No pre-emption
- Circular wait

4. What is a resource-allocation graph?

Resource allocation graph is directed graph which is used to describe deadlocks. 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 the set consisting of all active processes in the system and R the set consisting of all resource types in the system.

5. Define request edge and assignment edge.

A directed edge from process Pi to resource type Rj (denoted by Pi → Rj) is called as request edge; it signifies that process Pi requested an instance of resource type Rj and is currently waiting for that resource. A directed edge from resource type Rj to process Pi (denoted by Rj → Pi) is called an assignment edge; it signifies that an instance of resource type has been allocated to a process Pi.

6. What are the methods for handling deadlocks?

The deadlock problem can be dealt with in one of the three ways:

1. Use a protocol to prevent or avoid deadlocks, ensuring that the system will never enter a deadlock state.
2. Allow the system to enter the deadlock state, detect it and then recover.
3. Ignore the problem all together, and pretend that deadlocks never occur in the system.

REVIEW QUESTIONS AND ANSWERS

1. Suppose a spooling system has a saturation threshold of 75 percent and limits the maximum size of each file to 25 percent of the total spooling file size. Could deadlock occur in this system?

Ans: Yes, deadlock can still occur in this system. For instance, several jobs can begin transferring their outputs. When the spooling file reaches the 75 percent threshold, new jobs are not allowed. However, jobs that have begun are allowed to continue spooling, which may result in deadlock if there is insufficient space in the spooling file.

2. Suppose a spooling system has a saturation threshold of 75 percent and limits the maximum size of each file to 25 percent of the total spooling file size. Describe a simple way to ensure that deadlock will never occur in the system. Explain how this could lead to inefficient resource allocation.

Ans: A simple adjustment would be to allow only one job to continue spooling data when the file reaches the threshold. This would be inefficient because it would limit the maximum job size to much less than the available spooling space.

3. Describe how the four necessary conditions for deadlock apply to spooling systems.

Ans: No two jobs can simultaneously write data to the same location in the spooling file. Partially spooled jobs remain in the spooling file until more space is available. Jobs cannot remove other jobs from the spooling file. Finally, when the spooling file is full, each job waits for all of the other jobs to free up space.

4. Compare and contrast deadlock prevention and deadlock avoidance.

Ans: Deadlock prevention makes deadlock impossible but results in lower resource utilization. With deadlock avoidance, when the threat of deadlock approaches, it is sidestepped and resource utilization is higher. Systems using either deadlock prevention or deadlock avoidance will be free of deadlocks.

5. Some systems ignore the problem of deadlock. Discuss the costs and benefits of this approach.

Ans: Systems that ignore deadlock may fail when deadlock occurs. This is an unacceptable risk in mission-critical systems, but it may be appropriate in other systems where deadlocks rarely occur and the "cost" of dealing with an occasional deadlock is lower than the costs of implementing deadlock prevention or avoidance schemes.

6. (T/F) An unsafe state is a deadlocked state.

Ans: False. A process in an unsafe state might eventually deadlock, or it might complete its execution without entering deadlock. What makes the state unsafe is simply that the operating system cannot guarantee that from this state all processes can complete their work. From an unsafe state, it is possible but not guaranteed that all processes could complete their work, so a system in an unsafe state could eventually deadlock.

7. Describe the restrictions that the Banker's Algorithm places on processes.

Each process, before it runs, is required to specify the maximum number of resources it may require at any point during its execution. Each process cannot request more than the total number of resources in the system. Each process must also guarantee that once allocated a resource, the process will eventually return that resource to the system within a finite time.

8. Why is deadlock possible, but not guaranteed, when a system enters an unsafe state?

Ans: Processes could give back their resources early, increasing the number of available resources to the point that the state of the system was once again safe and all other processes could finish

9. Why does the Banker's Algorithm fail in systems that support hot swappable devices?

Ans: The Banker's Algorithm requires that the number of resources of each type remain fixed. Hot swappable devices can be added and removed from the system at any time, meaning that the number of resources of each type can vary.

10. Suppose a process has control of a resource of type R1. Does it matter which small circle points to the process in the resource-allocation graph?

Ans: No; all resources of the same type must provide identical functionality, so it does not matter which small circle within the circle R1 points to the process.

11. What necessary condition for deadlock is easier to identify in a resource-allocation graph than it is to locate by analyzing the resource-allocation data of all the system's processes?

Ans: Resource-allocation graphs make it easier to identify circular waits.

12. Why might deadlock detection be a better policy than either deadlock prevention or deadlock avoidance? Why might it be a worse policy?

Ans: In general, deadlock detection places fewer restrictions on resource allocation, thereby increasing resource utilization. However, it requires that the deadlock detection algorithm be performed regularly, which can incur significant overhead.

13. Suppose a system attempts to reduce deadlock detection overhead by performing deadlock detection only when there are a large number of processes in the system. What is one drawback to this strategy?

Ans: Because deadlock can occur between two processes, the system might not ever detect some deadlocks if the number of processes in the system is small.

KEY TERMS

circular wait—Condition for deadlock that occurs when two or more processes are locked in a "circular chain," in which each process in the chain is waiting for one or more resources that the next process in the chain is holding.

circular-wait necessary condition for deadlock—One of the four necessary conditions for deadlock; states that if a deadlock exists, there will be two or more processes in a circular chain such that each process is waiting for a resource held by the next process in the chain.

deadline scheduling—Scheduling a process or thread to complete by a definite time; the priority of the process or thread may need to be increased as its completion deadline approaches.

deadlock—Situation in which a process or thread is waiting for an event that will never occur.

deadlock avoidance—Strategy that eliminates deadlock by allowing a system to approach deadlock, but ensuring that deadlock never occurs. Avoidance algorithms can achieve higher performance than deadlock prevention algorithms.

deadlock detection—Process of determining whether or not a system is deadlocked. Once detected, a deadlock can be removed from a system, typically resulting in loss of work.

deadlock prevention—Process of disallowing deadlock by eliminating one of the four necessary conditions for deadlock.

deadlock recovery—Process of removing a deadlock from a system. This can involve suspending a process temporarily (and preserving its work) or sometimes killing a process (thereby losing its work) and restarting it.

dedicated resource—Resource that may be used by only one process at a time. Also known as a serially reusable resource.

Banker's Algorithm—Deadlock avoidance algorithm that controls resource allocation based on the amount of resources owned by the system, the amount of resources owned by each process and the maximum amount of resources that the process will request during execution. Allows resources to be assigned to processes only when the allocation results in a safe state.

graph reduction—Altering a resource-allocation graph by removing a process if that process can complete. This also involves removing any arrows leading to the process (from the resources allocated to the process) or away from the process (to resources the process is requesting). A resource-allocation graph can be reduced by a process if all of that process's resource requests can be granted, enabling that process to run to completion and free its resources.

maximum need (Dijkstra's Banker's Algorithm) —Characteristic of a process in Dijkstra's Banker's Algorithm that describes the largest number of resources (of a particular type) the process will need during execution.

mutual exclusion necessary condition for deadlock—One of the four necessary conditions for deadlock; states that deadlock can occur only if processes cannot claim exclusive use of their resources.

necessary condition for deadlock—Condition that must be true for deadlock to occur. The four necessary conditions are the mutual exclusion condition, no-preemption condition, wait-for condition and circular-wait condition.

no-preemption necessary condition for deadlock—One of the four necessary conditions for deadlock; states that deadlock can occur only if resources cannot be forcibly removed from processes.

nonpreemptible resource—Resource that cannot be forcibly removed from a process, e.g., a tape drive. Such resources are the kind that can become involved in deadlock.

preemptible resource—Resource that may be removed from a process such as a processor or memory. Such resources cannot be involved in deadlock.

reentrant code—Code that cannot be changed while in use and therefore can be shared among processes and threads.

resource allocation graph—Graph that shows processes and resources in a system. An arrow pointing from a process to a resource indicates that the process is requesting the resource. An arrow pointing from a resource to a process indicates that the resource is allocated to the process. Such a graph helps determine if a deadlock exists and. If so, helps identify the processes and resources involved in the deadlock.

resource type —Grouping of resources that perform a common task.

safe state—State of a system in Dijkstra's Banker's Algorithm in which there exists a sequence of actions that will allow every process in the system to finish without the system becoming deadlocked.

shared resource—Resource that can be accessed by more than one process. **starvation—** Situation in which a thread waits for an event that might never occur, also called indefinite postponement.

sufficient conditions for deadlock—The four conditions mutual exclusion, no-preemption, wait-for and circular wait-which are necessary and sufficient for deadlock.

suspend/resume—Method of halting a process, saving its state, releasing its resources to other processes, then restoring its resources after the other processes have released them.

transaction—Atomic, mutually exclusive operation that either completes or is rolled back. Modifications to database entries are often performed as transactions to enable high performance and reduce the cost of deadlock recovery.

unsafe state—State of a system in Dijkstra's Banker's Algorithm that might eventually lead to deadlock because there might not be enough resources to allow any process to finish.

wait-for condition—One of the four necessary conditions for deadlock; states that deadlock can occur only if a processis allowed to wait for a resource while it holds another.

EXPLANATORY QUESTIONS

1. Define deadlock.
2. Give an example of a deadlock involving only a single process and a single resource.

3. Give an example of a simple resource deadlock involving three processes and three resources. Draw the appropriate resource – allocation graph
4. Define and discuss each of the following resource concepts.
    a. preemptible resource
    b. nonpreemptible resource
    c. shared resource
    d. dedicated resource
    e. reentrant code
    f. serially reusable code
    g. dynamic resource allocation
5. State the four necessary conditions for a deadlock to exist. Give a brief intuitive argument for the necessity of each individual condition.
6. Explain the intuitive appeal of deadlock avoidance over deadlock prevention.
7. The fact that a state is unsafe does not necessarily imply that the system will deadlock. Explain why this is true. Give an example of an unsafe state and show how all of the processes could complete without a deadlock occurring.
8. Dijkstra's Banker's Algorithm has a number of weaknesses that preclude its effective use in real systems. Comment on why each of the following restrictions may be considered a weakness in the Banker's Algorithm.
    a. The number of resources to be allocated remains fixed.
    b. The population of processes remains fixed.
    c. The operating system guarantees that resource
    requests will be serviced in a finite time.
    d. Users guarantee that they will return held resources
    within a finite time.
    e. Users must state maximum resource
9. In a system in which it is possible for a deadlock to occur, under what circumstances would you use a deadlock detection algorithm?
10. In the deadlock detection algorithm employing the technique of graph reductions, show that the order of the graph reductions does not matter, the same final state will result.

[*Hint:* No matter what the order, after each reduction, the available resource pool increases.]

11. Why is deadlock recovery such a difficult problem?

12. Why is it difficult to choose which processes to "flush" in deadlock recovery?

EXERCISE PROBLEMS AND SOLUTIONS

1. Consider three process, all arriving at time zero, with total execution time of 10, 20 and 30 units respectively. Each process spends the first 20% of execution time doing I/O, the next 70% of time doing computation, and the last 10% of time doing I/O again. The operating system uses a shortest remaining compute time first scheduling algorithm and schedules a new process either when the running process gets blocked on I/O or when the running process finishes its compute burst. Assume that all I/O operations can be overlapped as much as possible. For what percentage of does the CPU remain idle?

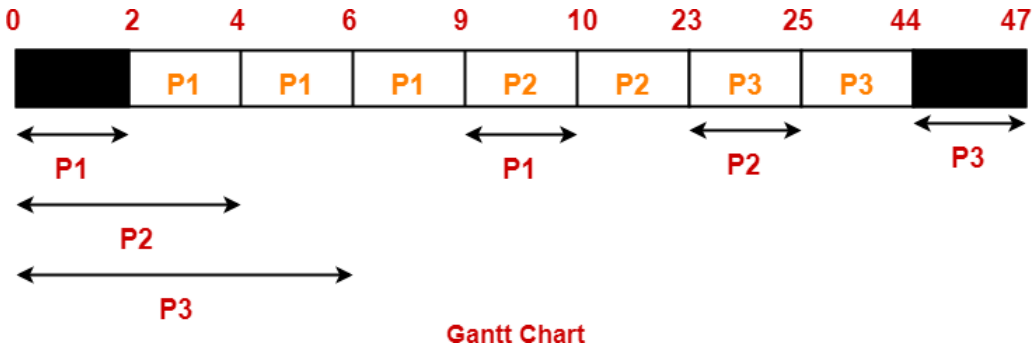
1.0% 2. 10.6% 3. 0.0% 4. 89.4%

Solution-

According to question, we have-

	Total Burst Time	I/O Burst	CPU Burst	I/O Burst
Process P1	10	2	7	1
Process P2	20	4	14	2
Process P3	30	6	21	3

Gantt Chart-



Gantt Chart

Percentage of time CPU remains idle

= (5 / 47) x 100

= 10.638%

Thus, Option (B) is correct.

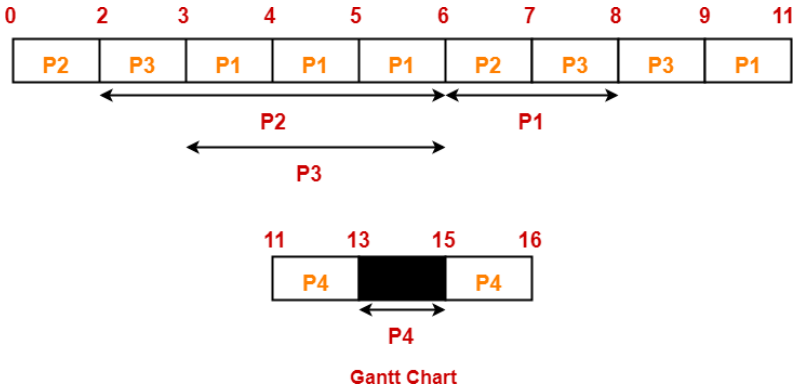
2. Consider the set of 4 processes whose arrival time and burst time are given below-

Process No.	***Arrival Time***	***Burst Time***		
		CPU Burst	***I/O Burst***	***CPU Burst***
P1	*0*	*3*	*2*	*2*
P2	*0*	*2*	*4*	*1*
P3	*2*	*1*	*3*	*2*
P4	*5*	*2*	*2*	*1*

If the CPU scheduling policy is Shortest Remaining Time First, calculate the average waiting time and average turn around time.

Solution

Gantt Chart



Gantt Chart

Now, we know-

- Turn Around time = Exit time – Arrival time
- Waiting time = Turn Around time – Burst time

Also read- Various Times Of Process

Process Id	Exit time	Turn Around time	Waiting time
P1	11	11 – 0 = 11	11 – (3+2) = 6
P2	7	7 – 0 = 7	7 – (2+1) = 4
P3	9	9 – 2 = 7	7 – (1+2) = 4
P4	16	16 – 5 = 11	11 – (2+1) = 8

- Average Turn Around time = (11 + 7 + 7 + 11) / 4 = 36 / 4 = 9 units
- Average waiting time = (6 + 4 + 4 + 8) / 4 = 22 / 5 = 4.4 units

3. Consider the set of 4 processes whose arrival time and burst time are given below-

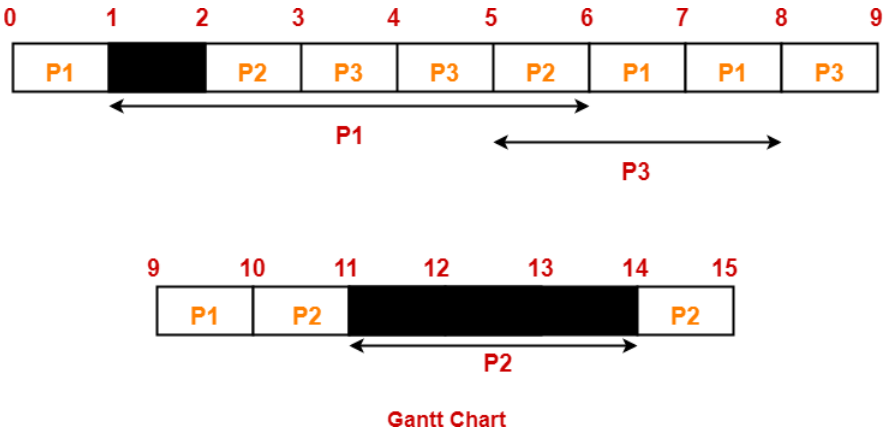
Process No.	***Arrival Time***	***Priority***	***Burst Time***		
			CPU Burst	***I/O Burst***	***CPU Burst***
P1	*0*	*2*	*1*	*5*	*3*
P2	*2*	*3*	*3*	*3*	*1*
P3	*3*	*1*	*2*	*3*	*1*

If the CPU scheduling policy is Priority Scheduling, calculate the average waiting time and average turn around time. (Lower number means higher priority)

Solution

The scheduling algorithm used is Priority Scheduling.

Gantt Chart



Gantt Chart

Now, we know-

- Turn Around time = Exit time – Arrival time
- Waiting time = Turn Around time – Burst time

Process Id	Exit time	Turn Around time	Waiting time
P1	10	10 – 0 = 10	10 – (1+3) = 6
P2	15	15 – 2 = 13	13 – (3+1) = 9
P3	9	9 – 3 = 6	6 – (2+1) = 3

Now,

- Average Turn Around time = (10 + 13 + 6) / 3 = 29 / 3 = 9.67 units
- Average waiting time = (6 + 9 + 3) / 3 = 18 / 3 = 6 units

4. *Consider three processes (process id 0, 1, 2 respectively) with compute time bursts 2, 4 and 8 time units. All processes arrive at time zero. Consider the longest remaining time first (LRTF) scheduling algorithm. In LRTF ties are broken by giving priority to the process with the lowest process id. The average turnaround time is:*

(A) 13 units **(B)** 14 units **(C)** 15units **(D)** 16units

Answer: (A)

Explanation: Let the processes be p0, p1 and p2. These processes will be executed in following order.

p2 p1 p2 p1 p2 p0 p1 p2 p0 p1 p2

0 4 5 6 7 8 9 10 11 12 13 14

Turn around time of a process is total time between submission of the process and its completion.

Turn around time of p0 = 12 (12-0)

Turn around time of p1 = 13 (13-0)

Turn around time of p2 = 14 (14-0)

Average turn around time is (12+13+14)/3 = 13.

5. *Consider three processes, all arriving at time zero, with total execution time of 10, 20 and 30 units, respectively. Each process spends the first 20% of execution time doing I/O, the next 70% of time doing computation, and the last 10% of time doing I/O again. The operating system uses a shortest remaining compute time first scheduling algorithm and schedules a new process either when the running process gets blocked on I/O or when the running process finishes its compute burst. Assume that all I/O operations can be overlapped as much as possible. For what percentage of time does the CPU remain idle?*

(A) 0% **(B)** 10.6% **(C)** 30.0% **(D)** 89.4%

Answer: (B)

Explanation: Let three processes be p0, p1 and p2. Their execution time is 10, 20 and 30 respectively. p0 spends first 2 time units in I/O, 7 units of CPU time and finally 1 unit in I/O.

p1 spends first 4 units in I/O, 14 units of CPU time and finally 2 units in I/O. p2 spends first 6 units in I/O, 21 units of CPU time and finally 3 units in I/O.

idle p0 p1 p2 idle

0 2 9 23 44 47

Total time spent = 47

Idle time = 2 + 3 = 5

Percentage of idle time = (5/47)*100 = 10.6 %

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

(A) 1 **(B)** 2 **(C)** 3 **(D)** 4

Answer: (B)

Explanation: Let three process be P0, P1 and P2 with arrival times 0, 2 and 6 respectively and CPU burst times 10, 20 and 30 respectively. At time 0, P0 is the only available process so it runs. At time 2, P1 arrives, but P0 has the shortest remaining time, so it continues. At time 6, P2 arrives, but P0 has the shortest remaining time, so it continues. At time 10, P1 is scheduled as it is the shortest remaining time process. At time 30, P2 is scheduled. Only two context switches are needed. P0 to P1 and P1 to P2.

7. *Which of the following process scheduling algorithm may lead to starvation*

(A) FIFO **(B)** Round Robin

(C) Shortest Job Next **(D)** None of the above

Answer: (C)

Explanation: Shortest job next may lead to process starvation for processes which will require a long time to complete if short processes are continually added.

8. *If the quantum time of round robin algorithm is very large, then it is equivalent to:*

(A) First in first out **(B)** Shortest Job Next

(C) Lottery scheduling **(D)** None of the above

Answer: (A)

Explanation: If time quantum is very large, then scheduling happens according to FCFS.

9. *Which of the following is FALSE about SJF (Shortest Job First Scheduling)?*

S1: It causes minimum average waiting time

S2: It can cause starvation

(A) Only S1 **(B)**Only S2

(C) Both S1 and S2 **(D)**Neither S1 nor S2

Answer: (D)

Explanation:

1. Both SJF and Shortest Remaining time first algorithms may cause starvation. Consider a situation when long process is there in ready queue and shorter processes keep coming.
2. SJF is optimal in terms of average waiting time for a given set of processes, but problems with SJF is how to know/predict time of next job.

10. *A single processor system has three resource types X, Y and Z, which are shared by three processes. There are 5 units of each resource type. Consider the following scenario, where the column alloc denotes the number of units of each resource type allocated to each process, and the column request denotes the number of units of each resource type requested by a process in order to complete execution. Which of these processes will finish LAST?*

1. P0
2. P1
3. P2
4. None of the above since the system is in a deadlock

	Alloc			**Request**		
	X	**Y**	**Z**	**X**	**Y**	**Z**
P0	1	2	1	1	0	3
P1	2	0	1	0	1	2
P2	2	2	1	1	2	0

Solution-

According to question-

- Total = [X Y Z] = [5 5 5]
- Total _Alloc = [X Y Z] = [5 4 3]

Now,

Available

= Total – Total_Alloc

= [5 5 5] – [5 4 3]

= [0 1 2]

Step-01:

- With the instances available currently, only the requirement of the process P1 can be satisfied.
- So, process P1 is allocated the requested resources.
- It completes its execution and then free up the instances of resources held by it.

Then,

Available

= [0 1 2] + [2 0 1]

= [2 1 3]

Step-02:

- With the instances available currently, only the requirement of the process P0 can be satisfied.
- So, process P0 is allocated the requested resources.
- It completes its execution and then free up the instances of resources held by it.

Then-

Available

= [2 1 3] + [1 2 1]

= [3 3 4]

Step-03:

- With the instances available currently, the requirement of the process P2 can be satisfied.
- So, process P2 is allocated the requested resources.
- It completes its execution and then free up the instances of resources held by it.

Then-

Available

= [3 3 4] + [2 2 1]

= [5 5 5]

Thus,

- There exists a safe sequence P1, P0, P2 in which all the processes can be executed.
- So, the system is in a safe state.
- Process P2 will be executed

at last. Thus, Option (C) is correct.

11. An operating system uses the banker's algorithm for deadlock avoidance when managing the allocation of three resource types X, Y and Z to three processes P0, P1 and P2. The table given below presents the current system state. Here, the Allocation matrix shows the current number of resources of each type allocated to each process and the Max matrix shows the maximum number of resources of each type required by each process during its execution.

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

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

REQ1: P0 requests 0 units of X, 0 units of Y and 2 units of Z

REQ2: P1 requests 2 units of X, 0 units of Y and 0 units of Z

Which of the following is TRUE?

1. Only REQ1 can be permitted
2. Only REQ2 can be permitted
3. Both REQ1 and REQ2 can be permitted
4. Neither REQ1 nor REQ2 can be permitted

Solution-

According to question,

Available = [X Y Z] = [3 2 2]

Now,

Need = Max – Allocation

So, we have-

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

Currently, the system is in safe state.

(It is given in question. If we want, we can check)

Checking Whether REQ1 Can Be Entertained-

- Need of P0 = [0 0 2]
- Available = [3 2 2]

Clearly,

- With the instances available currently, the requirement of REQ1 can be satisfied.
- So, banker's algorithm assumes that the request REQ1 is entertained.
- It then modifies its data structures as-

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

Available

= [3 2 2] – [0 0 2]

= [3 2 0]

- Now, it follows the safety algorithm to check whether this resulting state is a safe state or not.

- If it is a safe state, then REQ1 can be permitted otherwise not.

Step-01:

- With the instances available currently, only the requirement of the process P1 can be satisfied.
- So, process P1 is allocated the requested resources.
- It completes its execution and then free up the instances of resources held by it.

Then-

Available

= [3 2 0] + [3 2 0]

= [6 4 0]

Now,

- It is not possible to entertain any process.
- The system has entered the deadlock state which is an unsafe state.
- Thus, REQ1 will not be permitted.

Checking Whether REQ2 Can Be Entertained-

- Need of P1 = [2 0 0]
- Available = [3 2 2]

Clearly,

- With the instances available currently, the requirement of REQ1 can be satisfied.
- So, banker's algorithm assumes the request REQ2 is entertained.
- It then modifies its data structures as-

	Allocation			**Max**			**Need**		
	X	**Y**	**Z**	**X**	**Y**	**Z**	**X**	**Y**	**Z**
P0	0	0	1	8	4	3	8	4	2
P1	**5**	2	0	6	2	0	**1**	0	0
P2	2	1	1	3	3	3	1	2	2

Available

= [3 2 2] – [2 0 0]

= [1 2 2]

- Now, it follows the safety algorithm to check whether this resulting state is a safe state or not.

- If it is a safe state, then REQ2 can be permitted otherwise not.

Step-01:

- With the instances available currently, only the requirement of the process P1 can be satisfied.
- So, process P1 is allocated the requested resources.
- It completes its execution and then free up the instances of resources held by it.

Then-

Available

= [1 2 2] + [5 2 0]

= [6 4 2]

Step-02:

- With the instances available currently, only the requirement of the process P2 can be satisfied.
- So, process P2 is allocated the requested resources.
- It completes its execution and then free up the instances of resources held by it.

Then-

Available

= [6 4 2] + [2 1 1]

= [8 5 3]

Step-03:

With the instances available currently, the requirement of the process P0 can be satisfied.

- So, process P0 is allocated the requested resources.
- It completes its execution and then free up the instances of resources held by it.

Then-

Available

= [8 5 3] + [0 0 1]

= [8 5 4]

Thus,

- There exists a safe sequence P1, P2, P0 in which all the processes can be executed.
- So, the system is in a safe state.

- Thus, REQ2 can be permitted.

Thus, Correct Option is (B).

12. A system has 4 processes and 5 allocatable resource. The current allocation and maximum needs are as follows-

	Allocated					**Maximum**				
A	1	0	2	1	1	1	1	2	1	3
B	2	0	1	1	0	2	2	2	1	0
C	1	1	0	1	1	2	1	3	1	1
D	1	1	1	1	0	1	1	2	2	0

If Available = [0 0 X 1 1], what is the smallest value of x for which this is a safe state?

Solution-

Let us calculate the additional instances of each resource type needed by each process.

We know,

Need = Maximum – Allocation

So, we have-

	Need				
A	0	1	0	0	2
B	0	2	1	0	0
C	1	0	3	0	0
D	0	0	1	1	0

Case-01: For X = 0

If X = 0, then-

Available

= [0 0 0 1 1]

- With the instances available currently, the requirement of any process can not be satisfied.

- So, for X = 0, system remains in a deadlock which is an unsafe state.

Case-02: For X = 1

If X = 1, then-

Available

= [0 0 1 1 1]

Step-01:

- With the instances available currently, only the requirement of the process D can be satisfied.
- So, process D is allocated the requested resources.
- It completes its execution and then free up the instances of resources held by it.

Then-

Available

= [0 0 1 1 1] + [1 1 1 1 0]

= [1 1 2 2 1]

With the instances available currently, the requirement of any process can not be satisfied.

- So, for X = 1, system remains in a deadlock which is an unsafe state.

Case-02: For X = 2

If X = 2, then-

Available

= [0 0 2 1 1]

Step-01:

- With the instances available currently, only the requirement of the process D can be satisfied.
- So, process D is allocated the requested resources.
- It completes its execution and then free up the instances of resources held by it.

Then-

Available

= [0 0 2 1 1] + [1 1 1 1 0]

= [1 1 3 2 1]

Step-02:

- With the instances available currently, only the requirement of the process C can be satisfied.
- So, process C is allocated the requested resources.

- It completes its execution and then free up the instances of resources held by it.

Then-

Available

= [1 1 3 2 1] + [1 1 0 1 1]

= [2 2 3 3 2]

Step-03:

With the instances available currently, the requirement of both the processes A and B can be satisfied.

- So, processes A and B are allocated the requested resources one by one.
- They complete their execution and then free up the instances of resources held by it.

Then-

Available

= [2 2 3 3 2] + [1 0 2 1 1] + [2 0 1 1 0]

= [5 2 6 5 3]

Thus,

- There exists a safe sequence in which all the processes can be executed.
- So, the system is in a safe state.
- Thus, minimum value of X that ensures system is in safe state = 2.

UNIT 3. MEMORY MANAGEMENT

INTRODUCTION

- Memory accesses and memory management are a very important part of modern computer operation. Every instruction has to be fetched from memory before it can be executed, and most instructions involve retrieving data from memory or storing data in memory or both.
- The advent of multi-tasking OS compounds the complexity of memory management, as processes are swapped in and out of the CPU, so must their code and data be swapped in and out of memory, all at high speeds and without interfering with any other processes.
- Shared memory, virtual memory, the classification of memory as read-only versus read-write, and concepts like copy-on-write forking all further complicate the issue.

BASIC HARDWARE

From the memory chips point of view, all memory accesses are equivalent. The memory hardware doesn't know what a particular part of memory is being used for, nor does it care. This is almost true of the OS as well, although not entirely.

The CPU can only access its registers and main memory. It cannot, for example, make direct access to the hard drive, so any data stored there must first be transferred into the main memory chips before the CPU can work with it. (Device drivers communicate with their hardware via interrupts and "memory" accesses, sending short instructions for example to transfer data from the hard drive to a specified location in main memory. The disk controller monitors the bus for such instructions, transfers the data, and then notifies the CPU that the data is there with another interrupt, but the CPU never gets direct access to the disk.) Memory accesses to registers are very fast, generally one clock tick, and a CPU may be able to execute more than one machine instruction per clock tick. A memory buffer used to accommodate a speed differential, called a cache

Memory accesses to main memory are comparatively slow, and may take a number of clock ticks to complete. This would require intolerable waiting by the CPU if it were not for an intermediary fast memory cache built into most modern CPUs. The basic idea of the cache is to transfer chunks of memory at a time from the main memory to the cache, and then to access individual memory locations one at a time from the cache.

User processes must be restricted so that they only access memory locations that "belong" to that particular process. This is usually implemented using a base register and a limit register for each process, as shown in Figures 3.1 and 3.2 below.

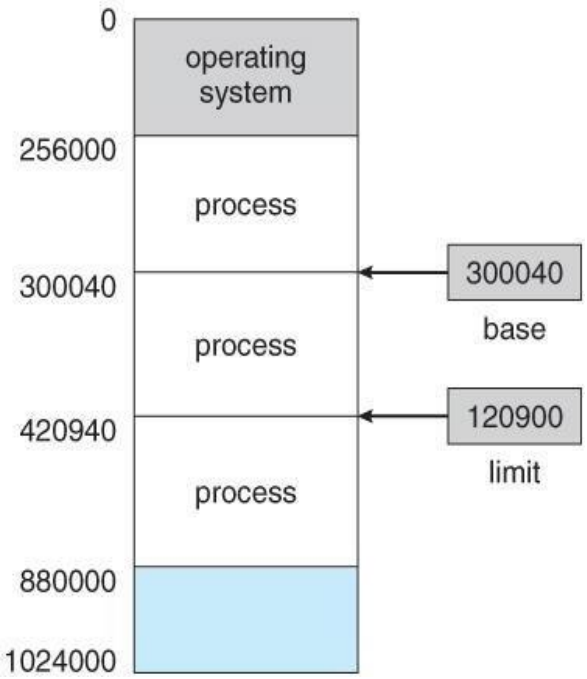


Figure 3.1 - A base and a limit register with a logical address space

Every memory access made by a user process is checked against these two registers, and if a memory access is attempted outside the valid range, then a fatal error is generated. The OS obviously has access to all existing memory locations, as this is necessary to swap users' code and data in and out of memory. It should also be obvious that changing the contents of the base and limit registers is a privileged activity, allowed only to the OS kernel.

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.

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 user memory. This provision allows the operating system to load users programs into user memory, to dump out those programs in case of errors, to access and modify parameters of system calls

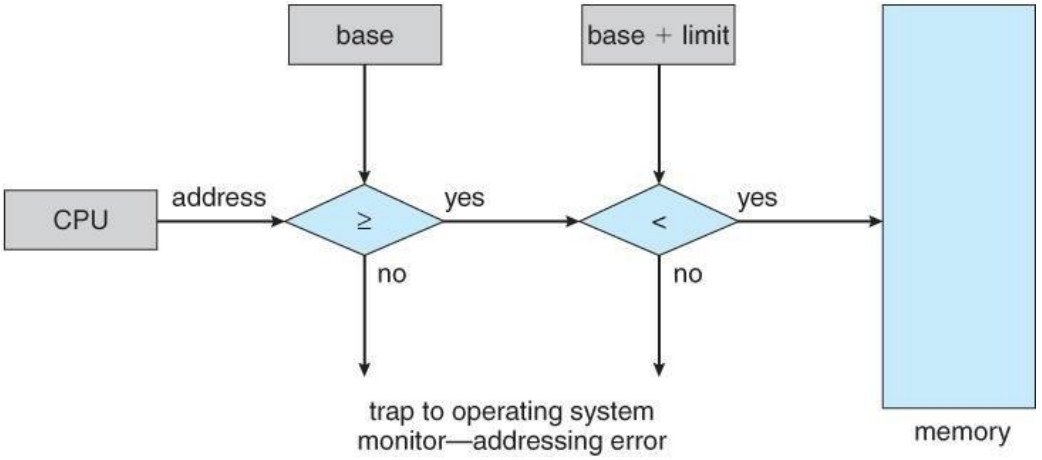


Figure 3.2 - Hardware address protection with base and limit registers

ADDRESS BINDING

User programs typically refer to memory addresses with symbolic names such as "i", "count", and "average Temperature". These symbolic names must be mapped or bound to physical memory addresses, which typically occurs in several stages:

- **Compile Time** - If it is known at compile time where a program will reside in physical memory, then absolute code can be generated by the compiler, containing actual physical addresses. However if the load address changes at some later time, then the program will have to be recompiled. DOS .COM programs use compile time binding.
- **Load Time** - If the location at which a program will be loaded is not known at compile time, then the compiler must generate relocatable code, which references addresses relative to the start of the program. If that starting address changes, then the program must be reloaded but not recompiled.
- **Execution Time** - If a program can be moved around in memory during the course of its execution, then binding must be delayed until execution time. This requires special hardware, and is the method implemented by most modern OS.

Figure 3.3 shows the various stages of the binding processes and the units involved in each stages.

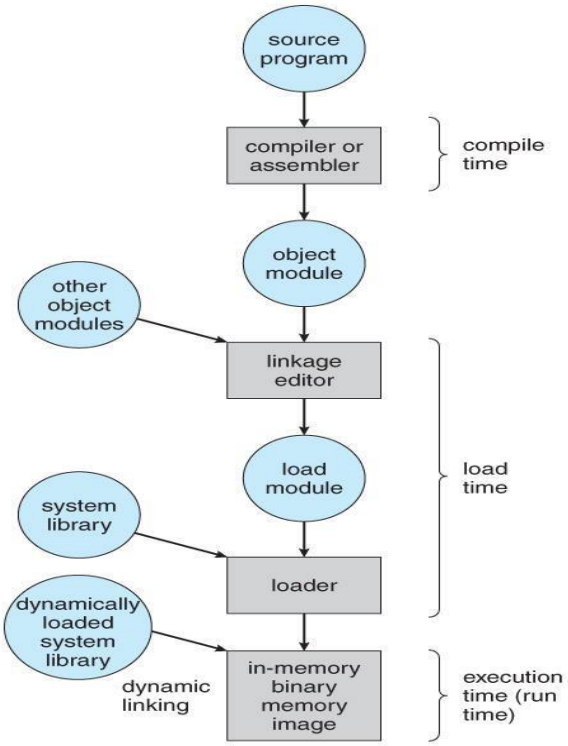


Figure 3.3 - Multistep processing of a user program

LOGICAL VERSUS 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 address binding scheme results in differing logical and physical addresses. In this case, we usually refer to the logical address as a **virtual address**. 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.

The address generated by the CPU is a logical address, whereas the address actually seen by the memory hardware is a physical address.

Addresses bound at compile time or load time have identical logical and physical addresses. Addresses created at execution time, however, have different logical and physical addresses.

The logical address is also known as a virtual address, and the two terms are used interchangeably by our text. The set of all logical addresses used by a program composes the logical address space, and the set of all corresponding physical addresses composes the physical address space. The run time mapping of logical to physical addresses is handled by the **Memory-Management Unit, MMU.**

The MMU can take on many forms. One of the simplest is a modification of the base-register scheme described earlier. The base register is now termed a relocation register, whose value is added to every memory request at the hardware level.

The user programs never uses physical addresses, User programs work entirely in logical address space, and any memory references or manipulations are done using purely logical addresses. Only when the address gets sent to the physical memory chips is the physical memory address generated.

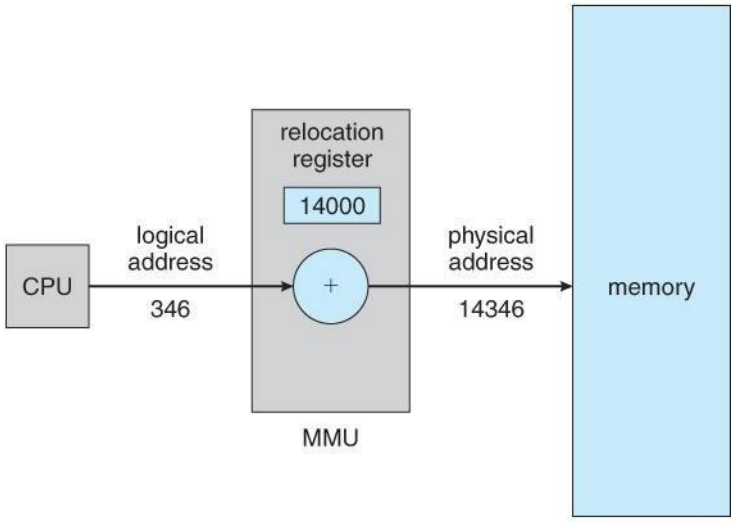


Figure 3.4 - Dynamic relocation using a relocation register

DYNAMIC LOADING

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.

Rather than loading an entire program into memory at once, dynamic loading loads up each routine as it is called. The advantage is that unused routines need never be loaded, reducing total memory usage and generating faster program start up times. The downside is the added complexity and overhead of checking to see if a routine is loaded every time it is called and then then loading it up if it is not already loaded.

The advantage of dynamic loading is that an unused routine is never loaded. This method is particularly useful when large amounts of code are needed to handle infrequently occurring cases, such as error routines. 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

Some operating systems support only static linking, in which system language 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 relinked 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.

Static linking library modules get fully included in executable modules, wasting both disk space and main memory usage, because every program that included a certain routine from the library would have to have their own copy of that routine linked into their executable code.

With dynamic linking, however, only a stub is linked into the executable module, containing references to the actual library module linked in at run time.

This method saves disk space, because the library routines do not need to be fully included in the executable modules, only the stubs. If the code section of the library routines is re-entrant, (meaning it does not modify the code while it runs, making it safe to re-enter it), then main memory can be saved by loading only one copy of dynamically linked routines into memory and sharing the code amongst all processes that are concurrently using it. (Each process would have their own copy of the data section of the routines, but that may be small relative to the code segments) that the OS must also manage shared routines in memory.

An added benefit of dynamically linked libraries (DLLs, also known as shared libraries or shared objects on UNIX systems) involves easy upgrades and updates. When a program uses a routine from a standard library and the routine changes, then the program must be re-built (re-linked) in order to incorporate the changes. If DLLs are used, then as long as the stub doesn't change, the program can be updated merely by loading new versions of the DLLs onto the system. Version information is maintained in both the program and the DLLs, so that a program can specify a particular version of the DLL if necessary.

The first time a program calls a DLL routine, the stub will recognize the fact and will replace itself with the actual routine from the DLL library. Further calls to the same routine will access the routine directly and not incur the overhead of the stub access.

SWAPPING

- A process must be loaded into memory in order to execute.
- If there is not enough memory available to keep all running processes in memory at the same time, then some processes who are not currently using the CPU may have their memory swapped out to a fast local disk called the ***backing store.***

The memory manager will start to swap out the process that just finished and to swap another process into the memory space that has been freed shown in Figure 3.5.

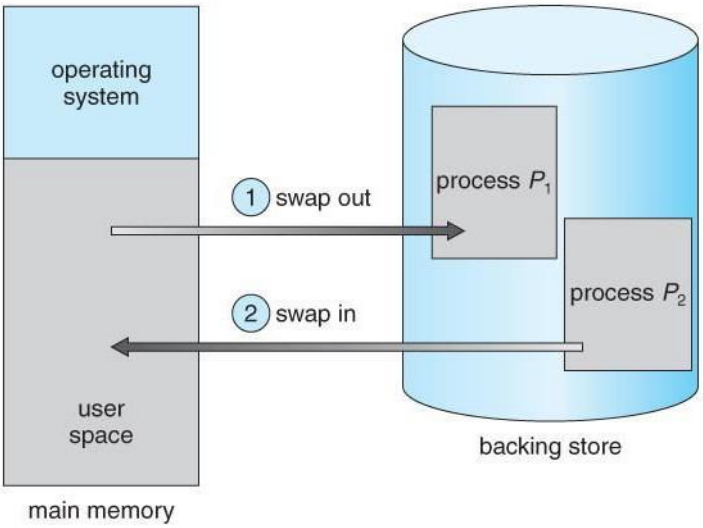


Figure 3.5 - Swapping of two processes using a disk as a backing store

A variant of this swapping policy is used for priority-based scheduling algorithms. If a higher-priority process arrives and wants service, the memory manager can swap out the lower-priority process and then load and execute the higher-priority process. When the higher-priority process finishes, the lower-priority process can be swapped back in and continued. This variant of swapping is sometimes called roll out, roll in. A process that is swapped out will be swapped back into the same memory space it occupied previously. This restriction is dictated by the method of address binding. If binding is done at assembly or load time, then the process cannot be easily moved to a different location. If execution-time binding is being used, however, then a process can be swapped into a different memory space, because the physical addresses are computed during execution time.

Swapping requires 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 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.

The context-switch time in such a swapping system is fairly high. To get an idea of the context switch time, let us 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.

STANDARD SWAPPING

If compile-time or load-time address binding is used, then processes must be swapped back into the same memory location from which they were swapped out. If execution time binding is used, then the processes can be swapped back into any available location.

Swapping is a very slow process compared to other operations. For example, if a user process occupied 10 MB and the transfer rate for the backing store were 40 MB per second, then it would take 1/4 second (250 milliseconds) just to do the data transfer. Adding in a latency lag of 8 milliseconds and ignoring head seek time for the moment, and further recognizing that swapping involves moving old data out as well as new data in, the overall transfer time required for this swap is 512 milliseconds, or over half a second. For efficient processor scheduling the CPU time slice should be significantly longer than this lost transfer time.

To reduce swapping transfer overhead, it is desired to transfer as little information as possible, which requires that the system know how much memory a process is using, as opposed to how much it might use. Programmers can help with this by freeing up dynamic memory that they are no longer using.

It is important to swap processes out of memory only when they are idle, or more to the point, only when there are no pending I/O operations. (Otherwise the pending I/O operation could write into the wrong process's memory space.) The solution is to either swap only totally idle processes, or do I/O operations only into and out of OS buffers, which are then transferred to or from process's main memory as a second step.

Most modern OSes no longer use swapping, because it is too slow and there are faster alternatives available. (e.g. Paging.) However some UNIX systems will still invoke swapping if the system gets extremely full, and then discontinue swapping when the load reduces again. Windows 3.1 would use a modified version of swapping that was somewhat controlled by the user, swapping process's out if necessary and then only swapping them back in when the user focused on that particular window.

SWAPPING ON MOBILE SYSTEMS

Swapping is typically not supported on mobile platforms, for several reasons: Mobile devices typically use flash memory in place of more spacious hard drives for persistent storage, so there is not as much space available.

- Flash memory can only be written to a limited number of times before it becomes unreliable.
- The bandwidth to flash memory is also lower.
- Apple's IOS asks applications to voluntarily free up memory
- Read-only data, e.g. code, is simply removed, and reloaded later if needed.
- Modified data, e.g. the stack, is never removed, but Apps that fail to free up sufficient memory can be removed by the OS
- Android follows a similar strategy.
- Prior to terminating a process, Android writes its application state to flash memory for quick restarting.

CONTIGUOUS MEMORY ALLOCATION

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. The memory is usually divided into two partitions:

- **Resident operating system**
- **User processes.**

The operating system can be placed either in 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.

Memory management is to load each process into a contiguous space. The operating system is allocated space first, usually at either low or high memory locations, and then the remaining available memory is allocated to processes as needed. (The OS is usually loaded low, because that is where the interrupt vectors are located, but on older systems part of the OS was loaded high to make more room in low memory (within the 640K barrier) for user processes.) In contiguous memory allocation, each process is contained in a single contiguous section of memory.

MEMORY PROTECTION

The system shown in Figure 3.6 below allows protection against user programs accessing areas that they should not, allows programs to be relocated to different memory starting addresses as needed, and allows the memory space devoted to the OS to grow or shrink dynamically as needs change.

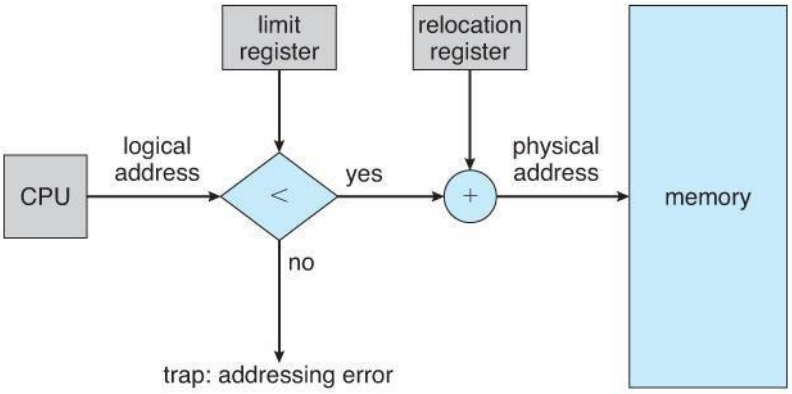


Figure 3.6 - Hardware support for relocation and limit registers

CONTIGUOUS MEMORY ALLOCATION

One method of allocating contiguous memory is to divide all available memory into equal sized partitions, and to assign each process to their own partition. This restricts both the number of simultaneous processes and the maximum size of each process, and is no longer used.

The memory blocks available comprise a set of holes of various sizes scattered throughout memory. When a process arrives and needs memory, the system searches the set for a hole that is large enough for this process. If the hole is too large, it is split into two parts. One part is allocated to the arriving process; the other is returned to the set of holes. When a process terminates, it releases its block of memory, which is then placed back in the set of holes. If the new hole is adjacent to other holes, these adjacent holes are merged to form one larger hole. At this point, the system may need to check whether there are processes waiting for memory and whether this newly freed and recombined memory could satisfy the demands of any of these waiting processes.

An alternate approach is to keep a list of unused (free) memory blocks (holes), and to find a hole of a suitable size whenever a process needs to be loaded into memory.

There are many different strategies for finding the "best" allocation of memory to processes, the three most common memory allocations are:

First fit - Search the list of holes until one is found that is big enough to satisfy the request, and assign a portion of that hole to that process. Whatever fraction of the hole not needed by the request is left on the free list as a smaller hole. Subsequent requests may start looking either from the beginning of the list or from the point at which this search ended.

Best fit - Allocate the smallest hole that is big enough to satisfy the request. This saves large holes for other process requests that may need them later, but the resulting unused portions of holes may be too small to be of any use, and will therefore be wasted. Keeping the free list sorted can speed up the process of finding the right hole.

Worst fit - Allocate the largest hole available, thereby increasing the likelihood that the remaining portion will be usable for satisfying future requests.

Simulations show that either first or best fit are better than worst fit in terms of both time and storage utilization. First and best fits are about equal in terms of storage utilization, but first fit is faster.

FRAGMENTATION

"**Fragmentation** occurs in a dynamic memory allocation system when many of the free blocks are too small to satisfy any request".

Fragmentation refers to the condition of a disk in which files are divided into pieces scattered around the disk. Fragmentation occurs naturally when you use a disk frequently, creating, deleting, and modifying files. At some point, the operating system needs to store parts of a file in non-contiguous clusters.

All the memory allocation strategies suffer from external fragmentation, though first and best fits experience the problems more so than worst fit. External fragmentation means that the available memory is broken up into lots of little pieces, none of which is big enough to satisfy the next memory requirement, although the sum total could.

The amount of memory lost to fragmentation may vary with algorithm, usage patterns, and some design decisions such as which end of a hole to allocate and which end to save on the free list.

Statistical analysis of first fit, for example, shows that for N blocks of allocated memory, another 0.5 N will be lost to fragmentation. There are two types of fragmentations they are:

- **Internal fragmentation**
- **External fragmentation**

External Fragmentation happens when a dynamic memory allocation algorithm allocates some memory and a small piece is left over that cannot be effectively used. If too much external fragmentation occurs, the amount of usable memory is drastically reduced. Total memory space exists to satisfy a request, but it is not contiguous.

Internal fragmentation occurs, with all memory allocation strategies. This is caused by the fact that memory is allocated in blocks of a fixed size, whereas the actual memory needed will rarely be that exact size. For a random distribution of memory requests, on the average 1/2 block will be wasted per memory request, because on the average the last allocated block will be only half full.

Note that the same effect happens with hard drives, and that modern hardware gives us increasingly larger drives and memory at the expense of ever larger block sizes, which translates to more memory lost to internal fragmentation.

Some systems use variable size blocks to minimize losses due to internal fragmentation.

If the programs in memory are relocatable, (using execution-time address binding).

Both the first-fit and best-fit strategies for memory allocation suffer from external fragmentation. The external fragmentation problem can be reduced via compaction, i.e. moving all processes down to one end of physical memory. This only involves updating the relocation register for each process, as all internal work is done using logical addresses.

Another solution as we will see in upcoming sections is to allow processes to use non-contiguous blocks of physical memory, with a separate relocation register for each block.

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.

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 such memory is available. And it can be solved using two techniques such as

- **Paging**
- **Segmentation**

Table 3.1 Comparison Chart between internal fragmentation and external fragmentation

BASIS FOR COMPARISON	INTERNAL FRAGMENTATION	EXTERNAL FRAGMENTATION
Basic	**It occurs when fixed sized memory blocks are allocated to the processes.**	**It occurs when variable size memory space are allocated to the processes dynamically.**
Occurrence	**When the memory assigned to the process is slightly larger than the memory requested by the process this creates free space in the allocated block causing internal fragmentation.**	**When the process is removed from the memory, it creates the free space in the memory causing external fragmentation.**
Solution	**The memory must be partitioned into variable sized blocks and assign the best fit block to the process.**	**Compaction, paging and segmentation.**

The problem of internal fragmentation can be reduced, but it cannot be totally eliminated. The paging and segmentation help in utilising the space freed due to external fragmentation by allowing a process to occupy the memory in a non-contiguous manner.

COMPACTION

The use of compaction is to minimize the probability of external fragmentation. In compaction, all the free partitions are made contiguous and all the loaded partitions are brought together.

By applying this technique, we can store the bigger processes in the memory. The free partitions are merged which can now be allocated according to the needs of new processes. This technique is also called defragmentation.

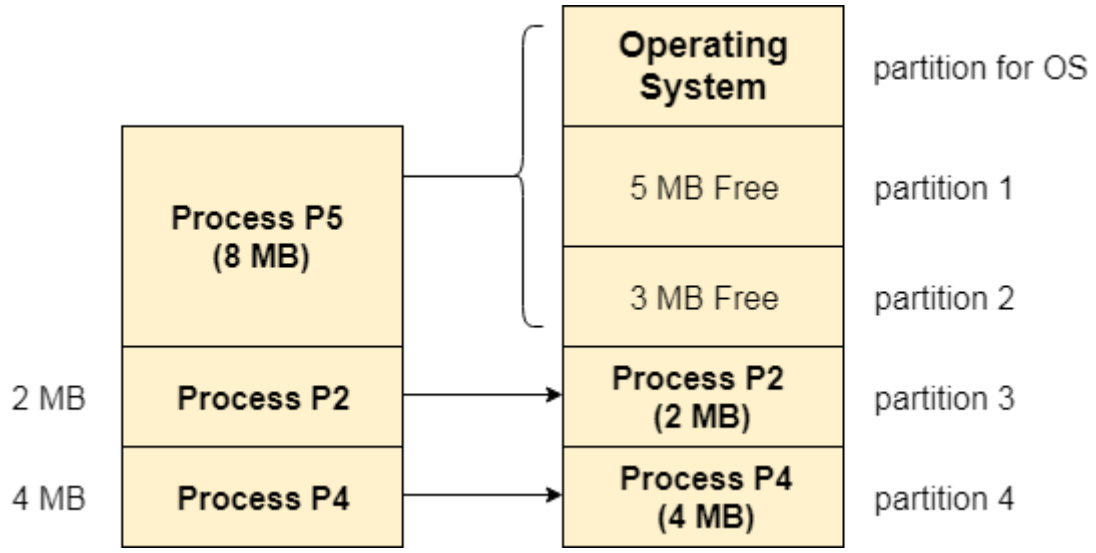


Figure 3.7 Compaction

As shown in the Figure 3.7, the process P5, which could not be loaded into the memory due to the lack of contiguous space, can be loaded now in the memory since the free partitions are made contiguous.

Problem with Compaction

The efficiency of the system is decreased in the case of compaction due to the fact that all the free spaces will be transferred from several places to a single place. Huge amount of time is invested for this procedure and the CPU will remain idle for all this time. Despite of the fact that the compaction avoids external fragmentation, it makes system inefficient.

TWO MARK QUESTIONS WITH ANSWERS

1. **Why page are sizes always powers of 2?**

   Ans: The paging is implemented by breaking up an address into a page and offset number. It is most efficient to break the address into X page bits and Y offset bits, rather than perform arithmetic on the address to calculate the page number and offset. Because each bit 25 26 position represents a power of 2, splitting an address between bits results in a page size that is a power of 2.

2. **Consider a logical address space of eight pages of 1024 words each, mapped onto a physical memory of 32 frames.**

   a. How many bits are there in the logical address?

   b. How many bits are there in the physical address?

Ans: Each page/frame holds 1K; we will need 10 bits to uniquely address each of those 1024 addresses. Physical memory has 32 frames and we need 25 bits to address each frame, requiring in total 5+10=15 bits. A logical address space of 64 pages requires 6 bits to address each page uniquely, requiring 16bits in total.

a. Logical address: 13 bits

b. Physical address: 15 bits

3. **In the IBM/370, memory protection is provided through the use of keys. A key is a 4-bit quantity. Each 2K block of memory has a key (the storage key) associated with it? The CPU also has a key (the protection key) associated with it. A store operation is allowed only if both keys are equal, or if either is zero. Which of the following memory-management schemes could be used successfully with this hardware?**

   - **Bare machine**
   - **Single-user system**
   - **Multiprogramming with a fixed number of processes**
   - **Multiprogramming with a variable number of processes**
   - **Paging**
   - **Segmentation**

Ans:

a) Protection not necessary set system key to 0.
b) Set system key to 0 when in supervisor mode.
c) Region sizes must be fixed in increments of 2k bytes, allocate key with memory blocks.
d) Same as above.
e) Frame sizes must be in increments of 2k bytes, allocate key with pages.
f) Segment sizes must be in increments of 2k bytes, allocate key with segments

4. **What is address binding?**

   Ans: The process of associating program instructions and data to physical memory addresses is called address binding, or relocation.

5. **Difference between internal and external fragmentation.**

   Ans: Internal fragmentation is the area occupied by a process but cannot be used by the process. This space is unusable by the system until the process release the space.

External fragmentation exists when total free memory is enough for the new process but it's not contiguous and can't satisfy the request. Storage is fragmented into small holes.

6. **Explain dynamic loading?**

   Ans: To obtain better memory-space utilization dynamic loading is used. 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 executed. If the routine needs another routine, the calling routine checks whether the routine has been loaded. If not, the relocatable linking loader is called to load the desired program into memory.

7. **Explain dynamic Linking.**

   Ans: Dynamic linking is similar to dynamic loading, rather that loading being postponed until execution time, linking is postponed. This feature is usually used with system libraries, such as language subroutine libraries. 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.

8. **Define swapping.**

   Ans: A process needs to be in memory to be executed. However a process can be swapped temporarily out of memory to a backing store and then brought back into memory for continued execution. This process is called swapping.

9. **Define lazy swapper.**

   Ans: Rather than swapping the entire process into main memory, a lazy swapper is used. A lazy swapper never swaps a page into memory unless that page will be needed.

10. **What are the common strategies to select a free hole from a set of available holes?**

    Ans: The most common strategies are,

    • First fit

    • Worst fit

    • Best fit

11. **Define effective access time.**

Ans: Let p be the probability of a page fault. The value of p is expected to be close to 0; that is, there will be only a few page faults. The effective access time is

Effective access time = (1-p) * ma + p * page fault time.

Where ma: memory-access time.

12. **How the problem of external fragmentation can be solved.**

    Ans: Solution to external fragmentation:

    a) Compaction: shuffling the fragmented memory into one contiguous location.

    b) Virtual memory addressing by using paging and segmentation.

13. **What you mean by compaction? In which situation is it applied.**

    Ans: Compaction is a process in which the free space is collected in a large memory chunk to make some space available for processes. In memory management, swapping Creates multiple fragments in the memory because of the processes moving in and out. Compaction refers to combining all the empty spaces together and processes.

14. **Define Address binding.**

    Ans: Address binding is the process of mapping the program's logical or virtual addresses to corresponding physical or main memory addresses. In other words, a given logical address is mapped by the MMU (Memory Management Unit) to a physical address.

15. **Define External Fragmentation.**

    Ans: It is a situation, when total memory available is enough to process a request but not in contiguous manner.

5 MARK QUESTIONS

1. Explain about the difference between internal fragmentation and external fragmentation.
2. What are the memory management requirements?
3. Explain difference between internal external fragmentations in detail.
4. Free memory holes of sizes 15K, 10K, 5K, 25K, 30K, 40K are available. The processes of size 12K, 2K, 25K, 20K is to be allocated. How processes are placed in first fit, best fit, worst fit. Calculate internal as well as external fragmentation.
5. Write short notes on swapping.

10 MARK QUESTIONS

1. Explain in detail about the concept of memory management.
2. Explain in detail about swapping and its techniques used.

KEY TERMS

Base register—Register containing the lowest memory address a process may reference.

Best-fit memory placement strategy—Memory placement strategy that places an incoming job in the smallest hole in memory that can hold the job.

Boundary register—Register for single-user operating systems that was used for memory protection by separating user memory space from kernel memory space.

Cache memory—Small, expensive, high-speed memory that holds copies of programs and data to decrease memory access times.

Coalescing memory holes—Process of merging adjacent holes in memory in variable partition multiprogramming systems. This helps create the largest possible holes available for incoming programs and data.

Contiguous memory allocation—Method of assigning memory such that all of the addresses in the process's entire address space are adjacent to one another.

Demand fetch strategy—Method of bringing program parts or data into main memory as they are requested by a process

Executive mode— protected mode in which a processor can execute operating system instructions on behalf of a user (also called kernel mode).

External fragmentation—Phenomenon in variable-partition memory systems in which there are holes distributed throughout memory that are too small to hold a process.

Fetch strategy—Method of determining when to obtain the next piece of program or data for transfer from secondary storage to main memory.

First-fit memory placement strategy—Memory placement strategy that places an incoming process in the first hole that is large enough to hold it.

Fixed-partition multiprogramming—Memory organization that divides main memory into a number of fixed-size partitions, each holding a single job.

Fragmentation (of main memory)—Phenomenon wherein a system is unable to make use of certain areas of available main memory.

Free memory list — Operating system data structure that points to available holes in memory.

SAMPLE PROBLEMS WITH SOLUTIONS

1. A computer has a single cache (off-chip) with a 2 ns hit time and a 98% hit rate. Main memory has a 40 ns access time. What is the computer's effective access time? If we add an on-chip cache with a .5 ns hit time and a 94% hit rate, what is the computer's effective access time? How much of a speedup does the on-chip cache give the computer?

   Answers:

   2 ns + .02 * 40 ns = 2.8 ns.

   With the on-chip cache, we have .5 ns + .06 * (2 ns + .02 * 40 ns) = .668 ns.

   The speedup is 2.8 / .668 = 4.2.

2. Assume a computer has on-chip and off-chip caches, main memory and virtual memory. Assume the following hit rates and access times: on-chip cache 95%, 1 ns, off-chip cache 99%, 10 ns, and main memory: X%, 50 ns, virtual memory: 100%, 2,500,000 ns. Notice that the on-chip access time is 1 ns. We do now want our effective access time to increase much beyond 1 ns. Assume that an acceptance effective access time is 1.6 ns. What should X be (the percentage of page faults) to ensure that EAT is no worse than 1.6 ns?

   Answer:

   EAT = 1ns + .05 * (10 ns + .01 * (50 ns + (1 – X) * 2,500,000 ns)).

   Since we want EAT to be no more than 1.6 ns,

   We solve for X with 1.6 ns = 1ns + .05 * (10 ns + .01 * (50 ns + (1 – X) * 2,500,000ns)).

   X = 1 – ((((((1.25 ns – 1 ns) / .05) – 10 ns) / .01) – 50 ns) / 2,500,000).

   X = 0.99994 = 99.994%.

   Our miss rate for virtual memory must be no worse than .006%!

UNIT 4 - SWAPPING

INTRODUCTION

A process must be in memory to be executed. A process, can be swapped temporarily out of memory to a backing store and then brought back into memory for continued execution. For example, assume a multiprogramming environment with a round-robin CPU-scheduling algorithm. When a quantum expires, the memory manager will start to swap out the process that has just finished and starts to swap another process into the memory space that has been freed (Figure1). Meanwhile, the CPU scheduler will allocate a timeslice to other process in memory. When each process finishes its quantum, it will be swapped with another process. A variant of this swapping policy is used for priority-based scheduling algorithms. If a higher-priority process arrives and wants service, the memory manager can swap out the lower-priority process and then load and execute the higher-priority process. When the higher-priority process finishes, the lower-priority process can be swapped back in and continued. This variant of swapping is sometimes called **roll out, roll in**. A process that is swapped out will be swapped back into the same memory space it occupied previously.

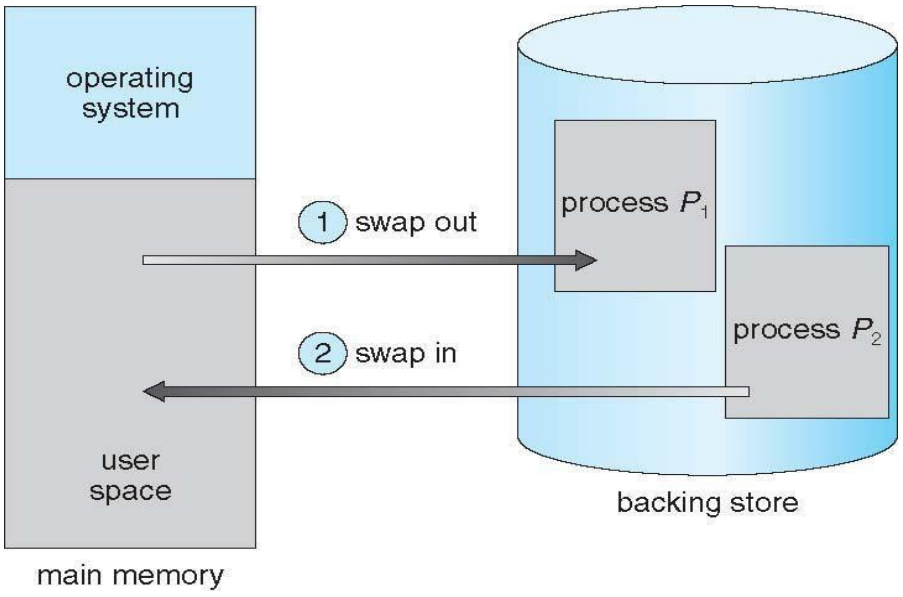


Figure 1: Swapping of two processes using a disk as a backing store

The system maintains a ready queue consisting of all processes whose memory images are on the backing store, a fast disk that is large enough to accommodate copies of all memory images for all users that 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 the next process is not in memory, 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. The context-switch time in such a swapping system is fairly high.

To get an idea of the context-switch time, let us 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. Assuming an average latency of 8 milliseconds, the swap time is 2008 milliseconds. Since we must both swap out and swap in, the total swap time is about 4016 milliseconds. Here, the major part of the swap time is transfer time. The total transfer time is directly proportional to the amount of memory swapped.

Generally, swap space is allocated as a chunk of disk, separate from the file system, so that its use is as fast as possible. Currently, standard swapping is used in few systems. It requires too much swapping time and provides too little execution time to be a reasonable memory-management solution.

MEMORY MANAGEMENT WITH BITMAPS

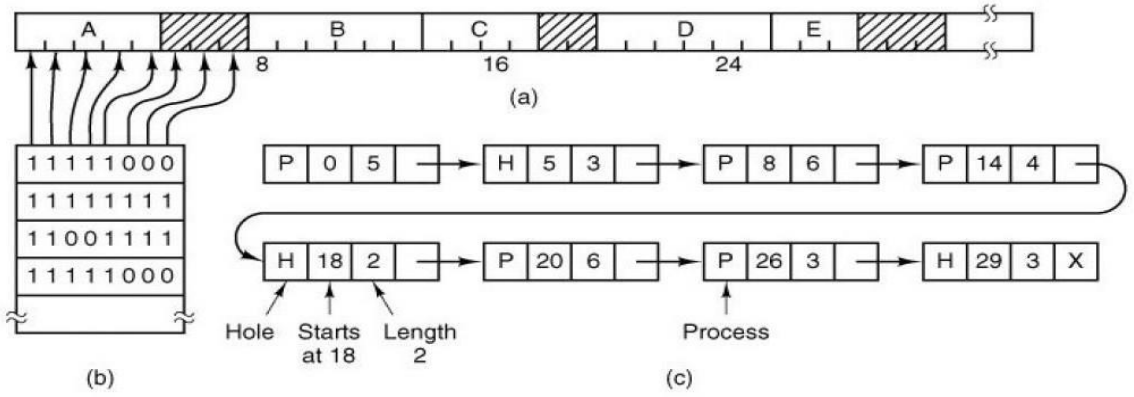


Figure 2: (a) A part of memory with five processes and three holes. The tick marks show the memory allocation units. The shaded regions (0 in the bitmap) are free. (b) The corresponding bitmap. (c) The same information as a list.

When memory is assigned dynamically, the operating system must manage it. With a bitmap. The memory is divided up into allocation units, perhaps as small as a few words and perhaps as large as several kilobytes. According to each allocation unit, a bit in the bitmap, which is 0 means the unit is free and 1 it is occupied (or vice versa). Figure 2 shows part of memory and the corresponding bitmap.

MEMORY MANAGEMENT WITH LINKED LISTS

Another way of keeping track of memory is to maintain a linked list of allocated and free memory segments, where a segment is either a process or a hole between two processes. In linked list each entry in the list specifies a hole (H) or process (P), the address at which it starts, the length, and a pointer to the next entry. Figure 3 gives an example, in which the segment list is kept sorted by address. Sorting this way has the advantage that when a process terminates or is swapped out, updating the list is straightforward.

A terminating process normally has two neighbours (except when it is at the very top or very bottom of memory). These may be either processes or holes, leading to the four combinations shown in figure 3. When the processes and holes are kept on a list sorted by address, several algorithms can be used to allocate memory for a newly created process (or an existing process being swapped in from disk). To allocate the memory for a process the following algorithms can be used.

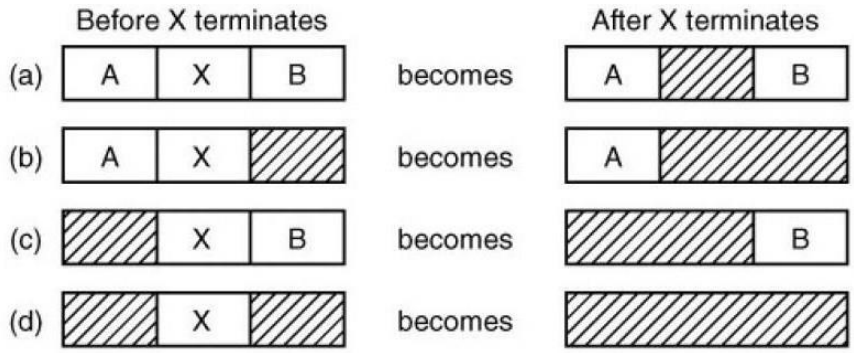


Figure 3: Four neighbour combinations for the terminating process, X.

First Fit: The simplest algorithm is first fit. The process manager scans along the list of segments until it finds a hole that is big enough to allocate the process. The hole is then broken

up into two pieces, one for the process and one for the unused memory, except in the statistically unlikely case of an exact fit. First fit is a fast algorithm because it searches as little as possible.

Next Fit: It works the same way as first fit, except that it keeps track of memory to find a suitable hole. The next time when it is called to find a hole, it starts searching the list from the place where it left the last time, instead of always at the beginning, as first fit does.

Best Fit: Best fit searches the entire list and takes the smallest hole that is adequate. Rather than breaking up a big hole that might be needed later, best fit tries to find a hole that is close to the actual size needed.

Worst Fit: Always take the largest available hole, so that the hole broken off will be big enough to be useful. Usually worst fit is not a very good idea because it takes a large amount of memory even for a small process.

Quick Fit: maintains separate lists for some of the more common sizes requested. For example, it might have a table with n entries, in which the first entry is a pointer to the head of a list of 4-KB holes, the second entry is a pointer to a list of 8-KB holes, the third entry a pointer to 12-KB holes, and so on. Holes of say, 21 KB, could either be put on the 20-KB list or on a special list of odd-sized holes. With quick fit, finding a hole of the required size is extremely fast, it has very less disadvantage of all other algorithms that sort by hole size, to finds its neighbours to see if it can a merge adjacent holes. which is expensive. If merging is not possible memory will quickly fragment into a large number of small holes into which no processes fit.

Both the first-fit and best-fit for memory allocation can suffer from **external fragmentation**. 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. Depending on the total amount of memory storage and the average process size, external fragmentation may be a minor or a major problem. 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, which leads to internal fragmentation which has unused memory that is internal to a partition.

One solution to the problem of external fragmentation is compaction. Compaction is not always possible, however. If relocation is static and is done at 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. 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 to physical memory wherever such memory is available. Two complementary techniques to achieve this solution: Paging and Segmentation, these techniques can also be combined.

PAGING

Paging is a memory-management scheme that permits the physical address space of a process to be non-contiguous. Paging avoids external fragmentation and the need for compaction. It also solves the considerable problem of fitting memory chunks of varying sizes onto the backing store. 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.

Definition: Paging is a memory-management scheme that permits the physical address space of a process to be non-contiguous. Paging avoids external fragmentation and the need for compaction.

MAPPING OF PAGES TO FRAMES

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 their source (a file system or 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.

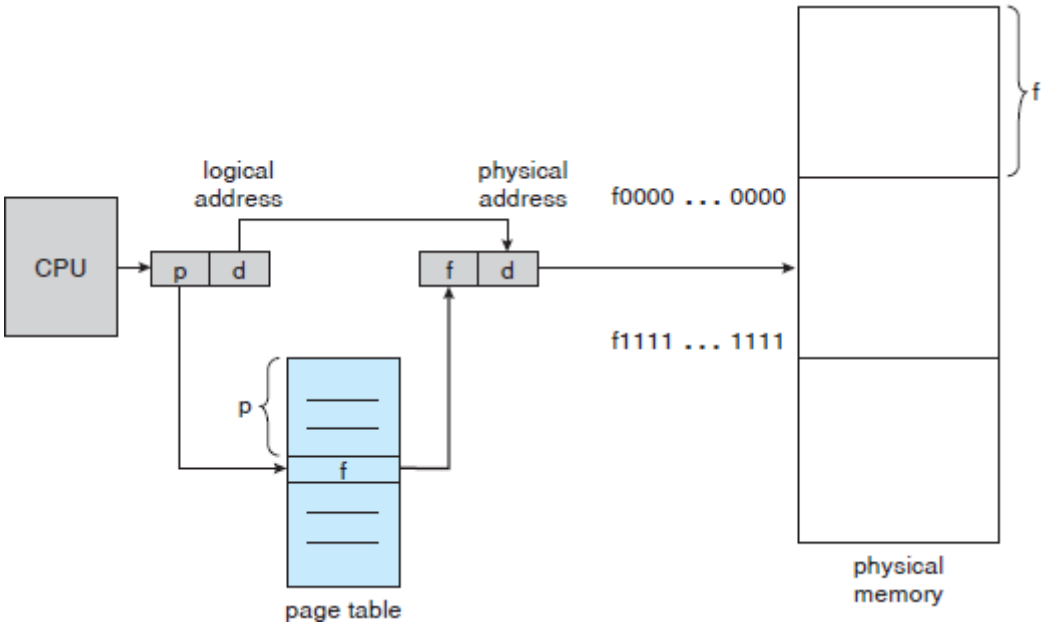


Figure 4: Paging hardware.

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. Figure 5 illustrates the paging model of memory.

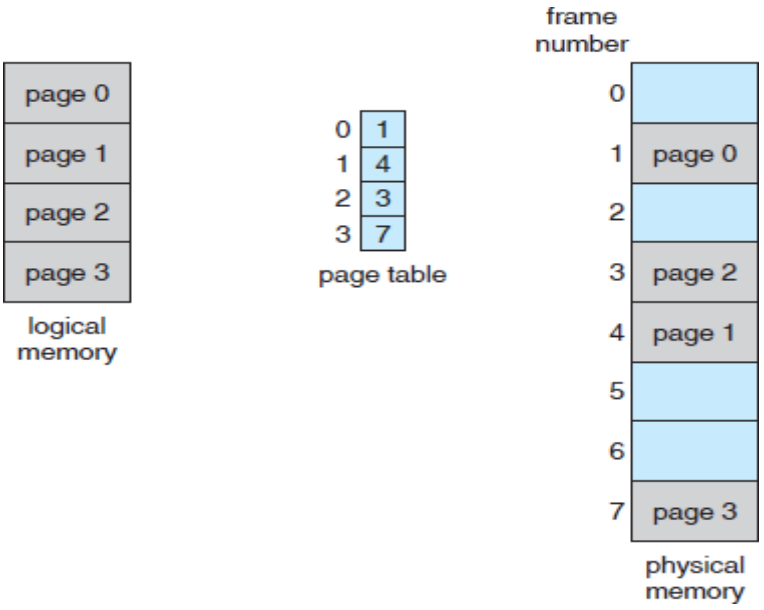
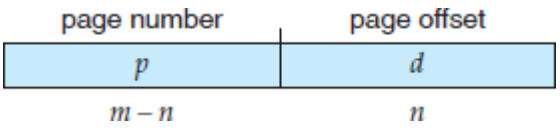


Figure 5: Paging Model of Logical and Physical Memory

The page size is defined by the hardware and 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. If the size of the logical address space is 2*m* and a page size is 2*n* 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:



For example, consider the memory in Figure 6, where the logical address has *n*= 2 and *m* = 4. Using a page size of 4 bytes and a physical memory of 32 bytes (8 pages), 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 also shows paging itself is a form of dynamic relocation and every logical address is bound by the paging hardware to some physical address.

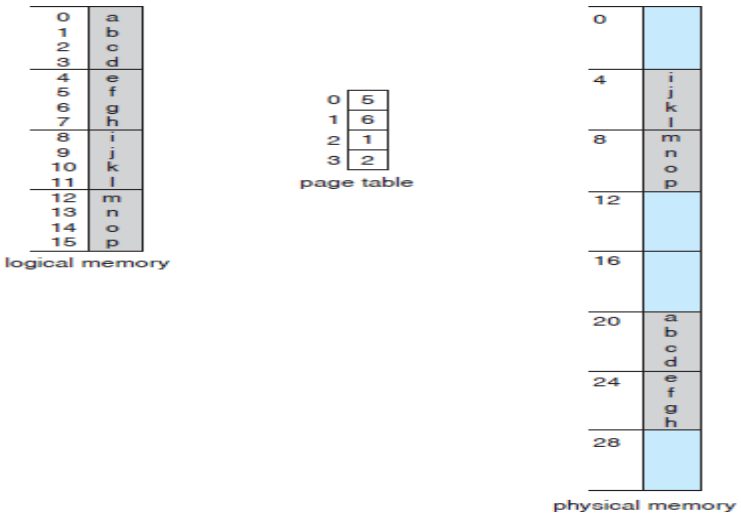


Figure 6: Paging example for a 32-byte memory with 4-byte pages

When we use a paging scheme, we have no external fragmentation, whereas if process size is independent of page size, there internal fragmentation can be expected to average one-half page per process. This 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. So, When a process arrives in the system to be executed, its size, expressed in pages, is examined, since 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, its frame number is put into the page table, and so on as shown in Figure 7.

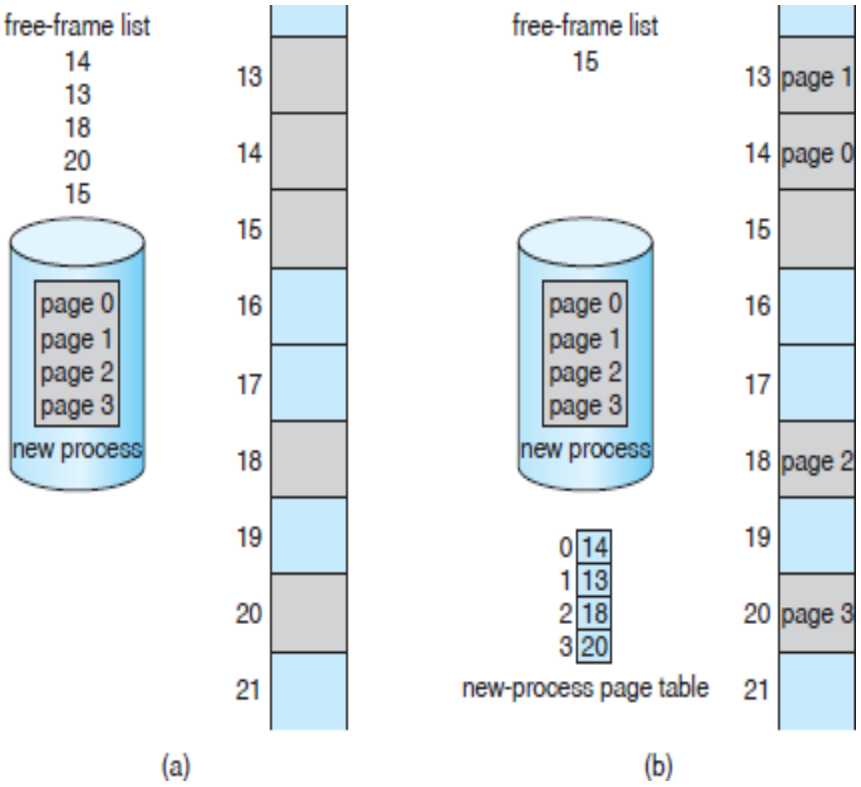


Figure 7: Free frames (a) before allocation and (b) after allocation.

In paging user program views memory as one single space containing only one program. But, 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. 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. Thus, 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 is also used by the CPU dispatcher to define the hardware page table when a process is to be allocated to the CPU. Paging therefore increases the context-switch time.

Hierarchical Page Table

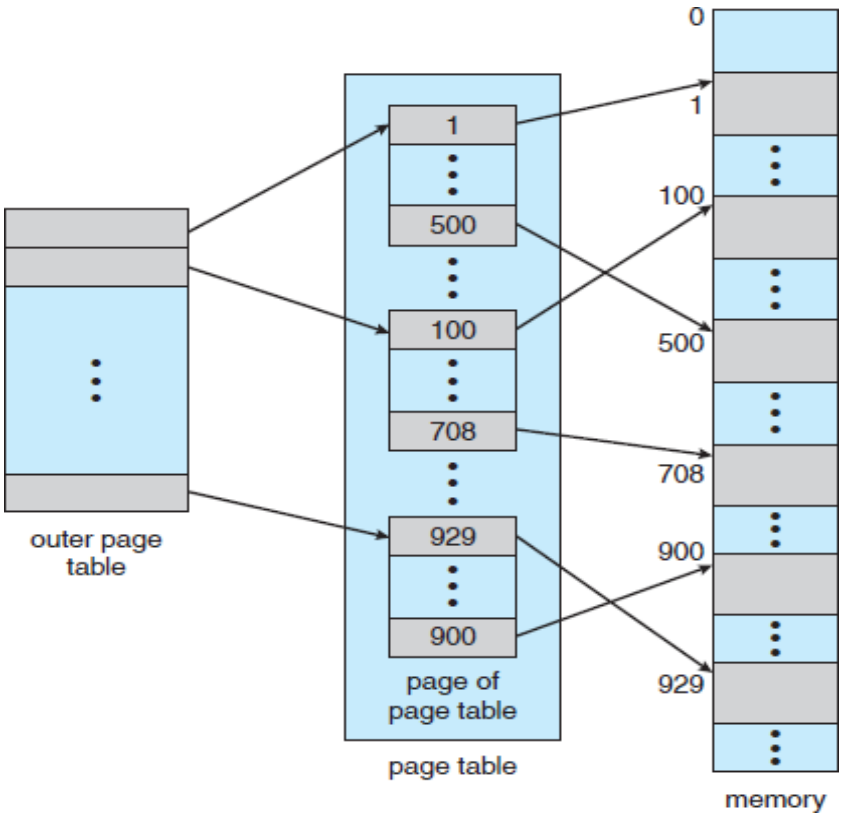
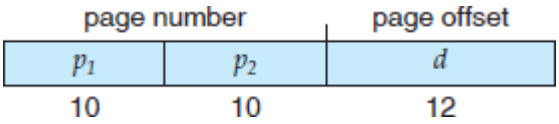


Figure 8: A Two-level Page-table Scheme.

The most common techniques for structuring the page table is Hierarchical Paging. Most modern computer systems support a large logical address space (232 to 264). In such an environment, the page table itself becomes excessively large. For example, consider a system with a 32-bit logical address space. If the page size in such a system is 4 KB (212), then a page table may consist of up to 1 million entries (232/212). Assuming that each entry consists of 4 bytes, each process may need up to 4 MB of physical address space for the page table alone. Clearly, we would not want to allocate the page table contiguously in main memory. One simple solution to this problem is to divide the page table into smaller pieces. One way is to use a two-level paging algorithm, in which the page table itself is also paged as Figure 8.

For example, consider the system with a 32-bit logical address space and a page size of 4 KB. A logical address divided into a page number consisting of 20 bits and a page offset consisting of 12 bits. Because we page the page table, the page number is further divided into a 10-bit page number and a 10-bit page offset. Thus, a logical address is as follows:



where $p1$ is an index into the outer page table and $p2$ is the displacement within the page of the inner page table. The address-translation method for this architecture is shown in Figure 9. Because address translation works from the outer page table inward, this scheme is also known as a forward-mapped page table.

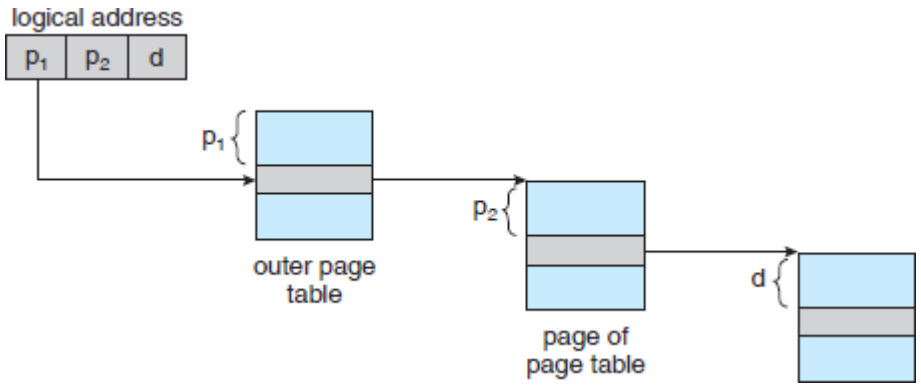
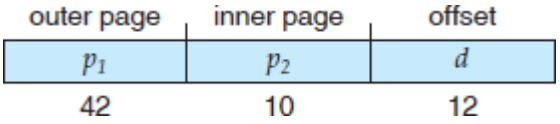
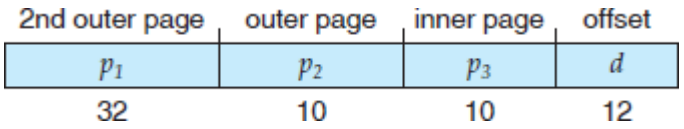


Figure 9: Address translation for a two-level 32-bit paging architecture.

For a system with a 64-bit logical address space, a two-level paging scheme is no longer appropriate. To illustrate this point, consider the page size with 4 KB (212). In this case, the page table has up to 252 entries. If we use a two-level paging scheme, then the inner page tables can conveniently be one page long, or contain 210 4-byte entries. The addresses look like this:



The outer page table consists of 242 entries, or 244 bytes. The way to avoid such a large table is to divide the outer page table into smaller pieces. We can divide the outer page table in various ways, we can page the outer page table, giving us a three-level paging scheme. Suppose that the outer page table is made up of standard-size pages (210 entries, or 212 bytes), the 64-bit address space is still daunting:



Thus he next step would be a four-level paging scheme, where the second-level outer page table itself is also paged, and so forth.

SEGMENTATION

Segmentation is a memory-management scheme that supports the 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. This in contrast 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, 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>

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

A C compiler might create separate segments for the following:

1. The code

2. Global variables

3. The heap, from which memory is allocated

4. The stacks used by each thread

5. 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.

Definition: Segmentation is a memory-management scheme that supports the user view of memory, where logical address space is a collection of segments. Each segment has a name and a leInngtshe.gTmheentaadtdiornestshees uspsecirfeyfebrosthtothoebjseecgtsmiennt neapmreogarnadmthbeyoafftsweot -wdiitmhiennsthioensaelgamdednret.ss,

whereas the actual physical memory is still, as one-dimensional sequence of bytes. Thus, we define an implementation to map two dimensional user-defined addresses into one-dimensional physical addresses. This mapping is effected 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, and the segment limit specifies the length of the segment. The use of a segment table is illustrated in Figure 9.

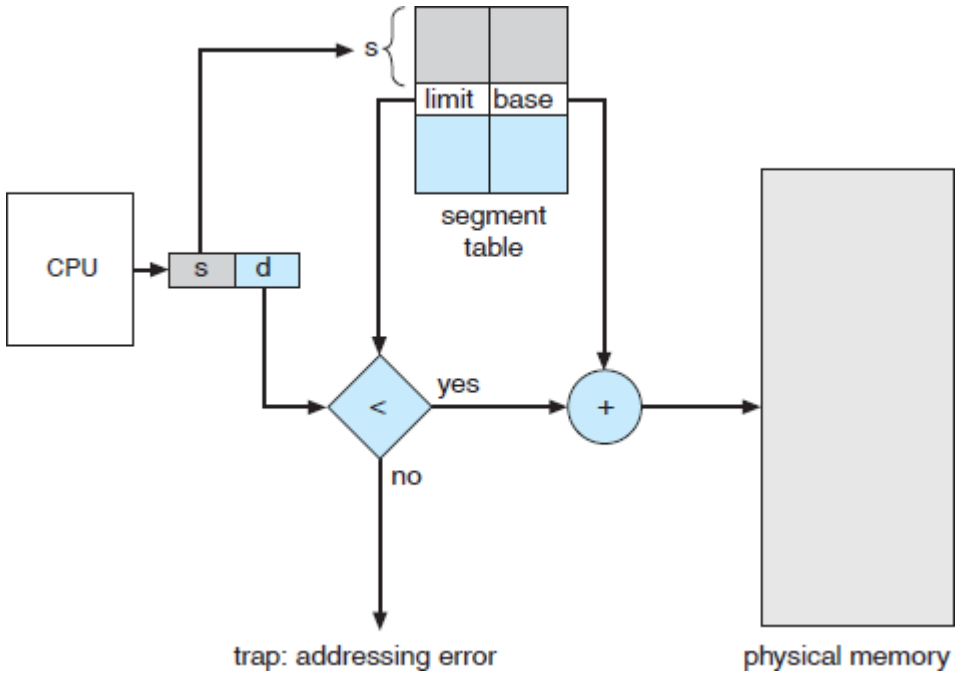


Figure 9: Segmentation Hardware

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 no within the limit the operating system concludes that the logical addressing attempts to trap 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. Thus the segment table is an array of base–limit register pairs. The Figure 10 shows five segments numbered from 0 through 4. The segments are stored in physical memory as shown Figure 10. 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.

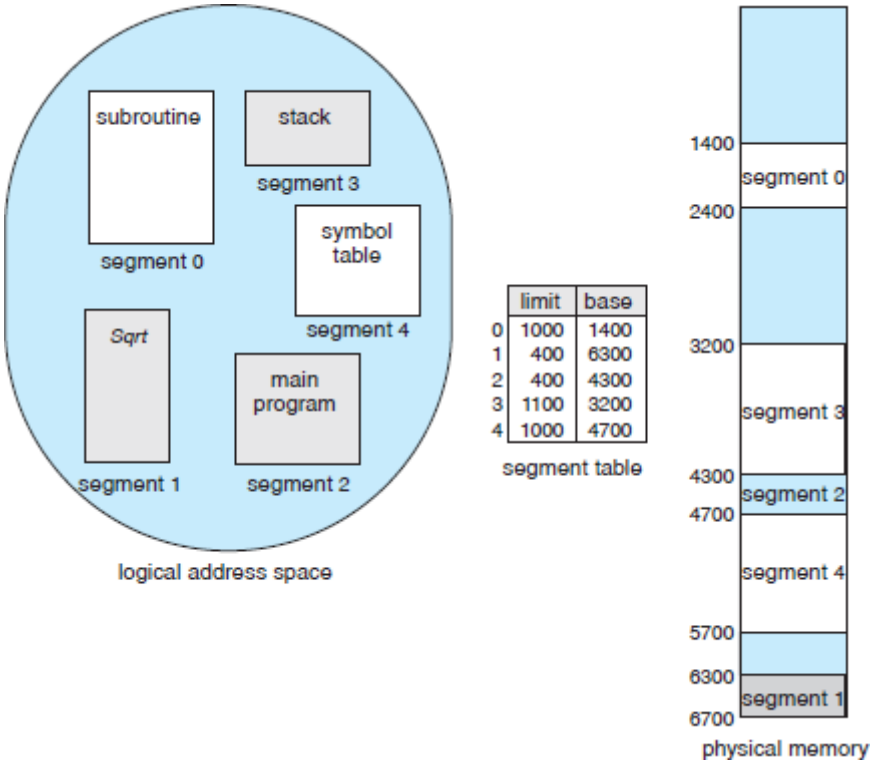


Figure 10: Example of Segmentation

Virtual Memory

The instructions which must be executed must be in physical memory. The ability to execute a program that is only partially in memory can have many benefits:

(i) A program will no longer need to have a constrain by the amount of physical memory that is available and also the users can write programs for an extremely large *virtual* address space, by simplifying the programming task.

(ii) Because each user program could take less physical memory, more programs could be run at the same time, with increase in CPU utilization and throughput. But this will not increase response time or turnaround time.

Definition: Virtual memory is a technique that allows the execution of processes that are not completely in memory. Virtual memory also allows processes to share files easily and to implement shared memory.

• Less I/O will be needed to load or swap user programs into memory, so that each user program will run faster. Thus, running a program that is not entirely in memory would benefit both the system and the user.

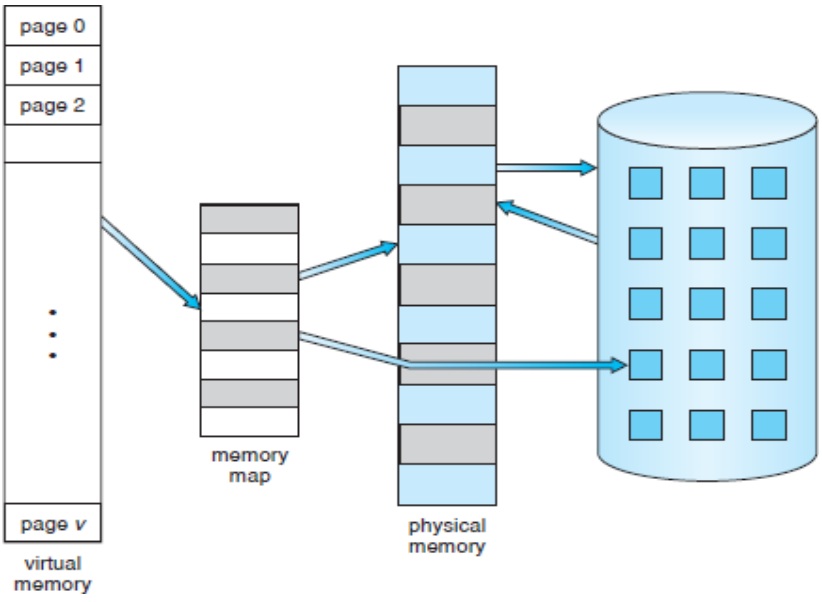


Figure 11: Virtual Memory that is Larger than Physical Memory

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 as shown in Figure 11. 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. This view of a process begins at a certain logical address—say, address 0—and exists in contiguous memory, as shown in Figure 12, where the physical memory is organized in page frames and the physical page frames that are assigned to a process may not be contiguous. In Figure 12 the heap is allowed to grow upward in memory as used for dynamic memory allocation. Similarly, the stack is allowed 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 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 during program execution.

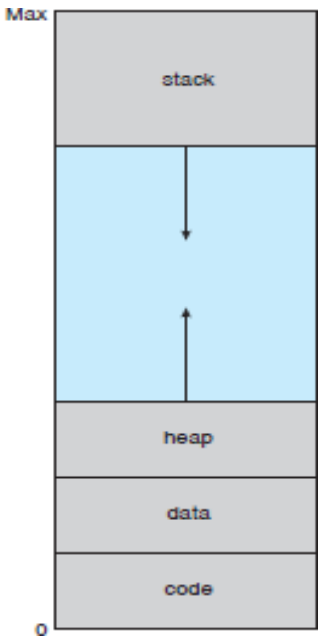


Figure 12: Virtual Address Space

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

(i) 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 as shown in Figure 13.

(ii) Similarly, virtual memory enables processes to share 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 as a part of their virtual address space, yet the actual physical pages of memory are shared, as illustrated in Figure 13.

(iii) Virtual memory also allows pages to be shared during process creation with the fork() system call, thus speeding up process creation.

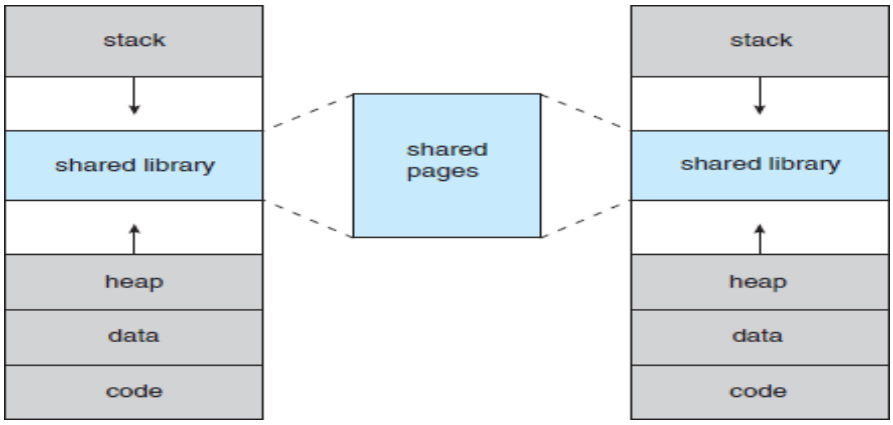


Figure 13: Shared library using virtual memory.

DEMAND PAGING

An executable program must be loaded from disk into memory. One option is to load the entire program in physical memory at program execution time. 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 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 loaded only when they are demanded during program execution. Pages that are never accessed are never loaded into the physical memory. A demand-paging system is similar to a paging system with swapping as shown in

Figure 14. When we want to execute a process, we swap it into memory. Rather than swapping the entire process into memory. This is known as **lazy swapper**.

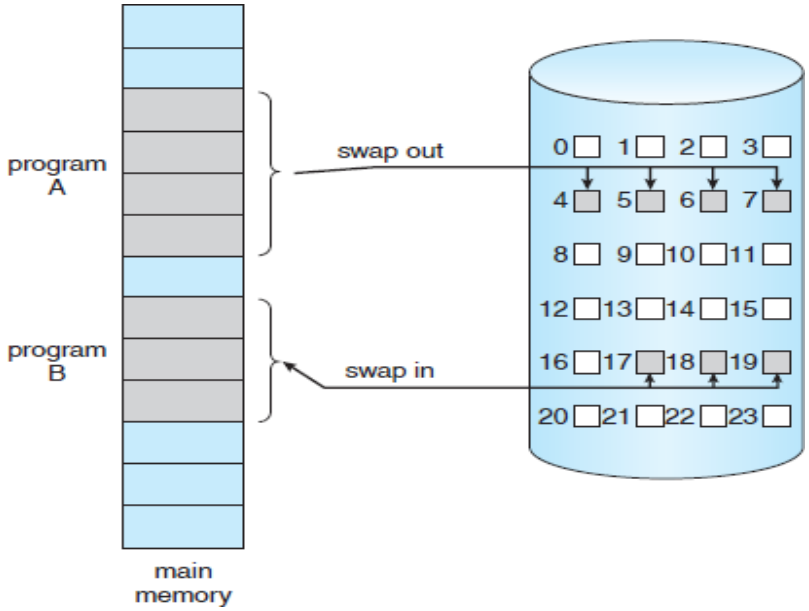
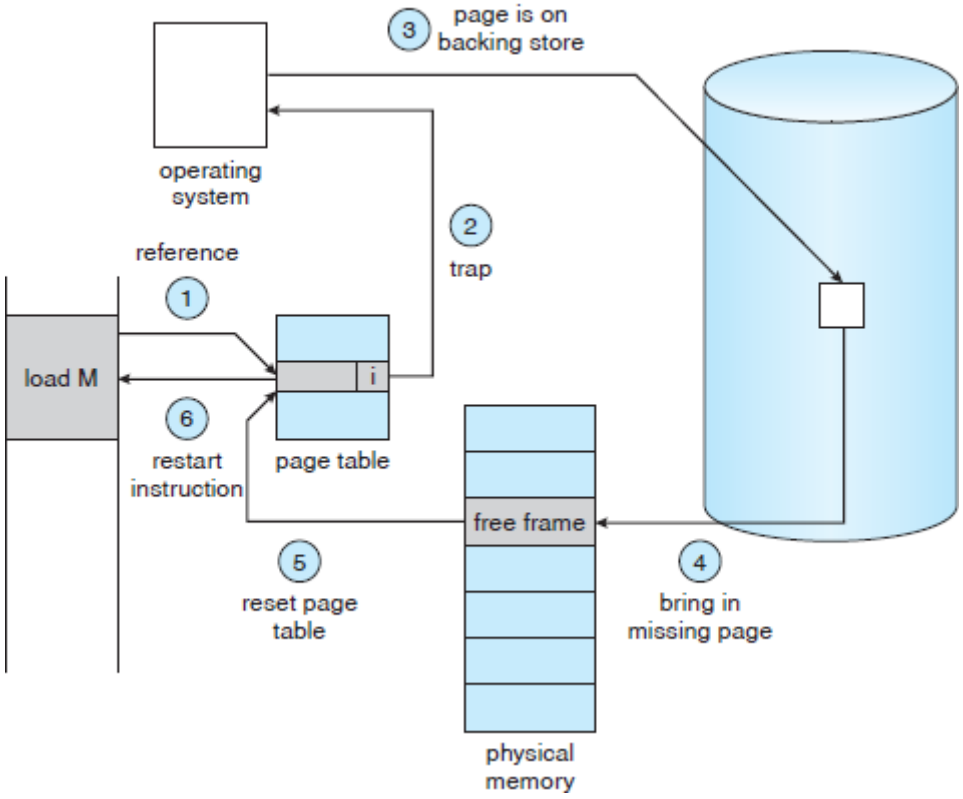


Figure 14: Transfer of a paged memory to contiguous disk space

A lazy swapper never swaps a page into memory unless that page will be needed. A swapper is used to manipulate the entire processes, whereas a **pager** is concerned with the individual pages of a process. Thus use *pager*, rather than *swapper*, in connection with demand paging. 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 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. For this purpose, the valid–invalid bit scheme is used. When this bit is set as "valid," the associated page is both legal and in memory. If the bit is set as "invalid," the page either is not valid 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 marked invalid or contains the address of the page on disk. This situation is depicted in Figure 15. Thus marking a page invalid will have no effect if the process never attempts to access that page. Hence, only those pages that are actually needed, will be brought in the memory and then 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.



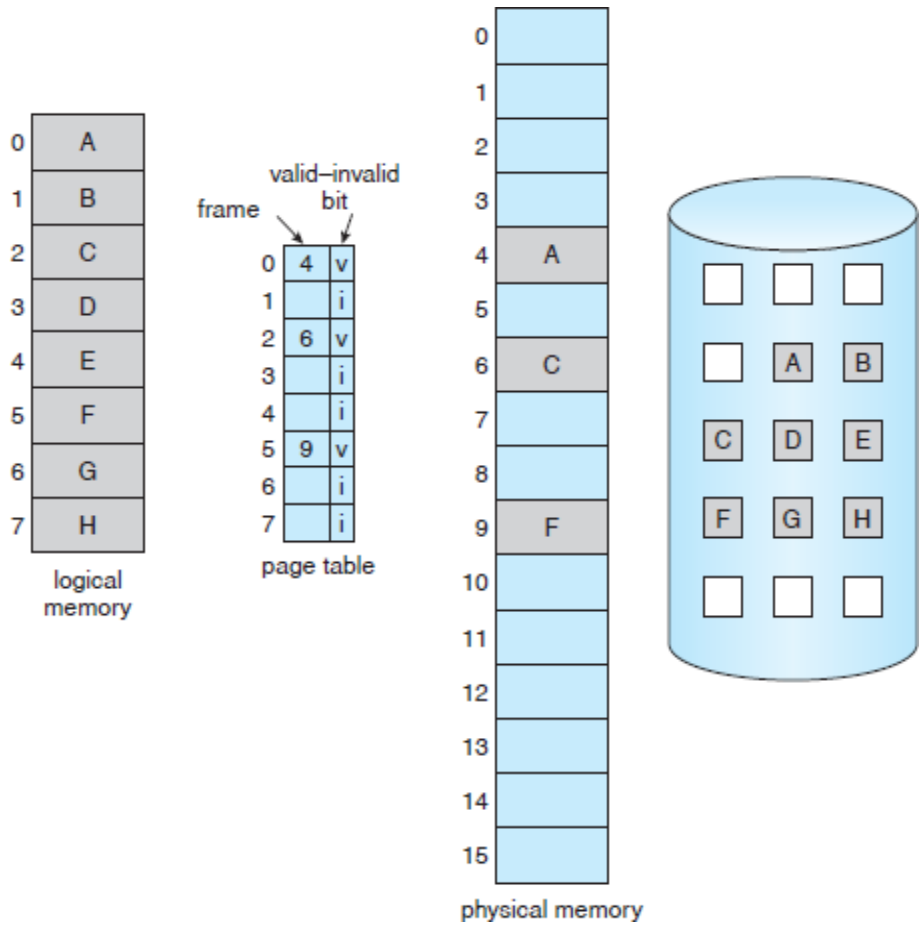


Figure 15: Page Table when some pages are not in main memory.

If the process tries to access a page that is not in the memory, access to that page is marked as invalid thus causing a page fault. 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.

Definition: Accessing the page that is not currently in memory for execution is called as Page Fault.

The figure 16 shows the procedure for handling this page fault as follows,

1. Checks an internal table usually kept with the process control block to determine whether the reference has a valid or an invalid memory access.

2. If the reference is invalid, the process will be terminate. If it is valid, and the page is not yet brought in the memory, then the page is brought in.

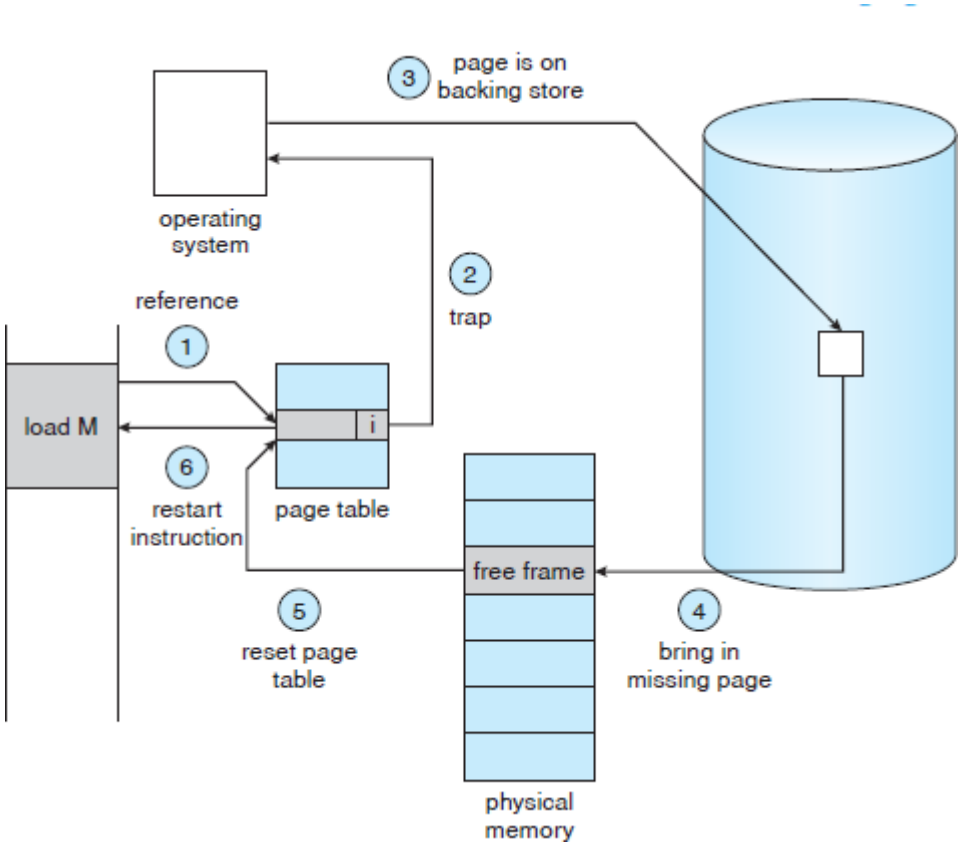


Figure 16: Steps in handling a page fault.

3. Finds a free frame by taking one from the free-frame list.
4. Schedules a disk operation to read the desired page into the newly allocated frame.
5. When the disk read is complete, the internal table is modified. The process and the page table is modified to indicate that the page is in memory.
6. The instruction that was interrupted by the trap is restarted. The process access the page as though it had always been in memory.

In the extreme case, the execution of a process starts 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 which the page is brought into memory, to continue the execution. At this point, process executes with no more faults. This scheme is **pure demand paging**, which never brings a page into memory until it is required.

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. As a worst-case example, consider a three-address instruction such as ADD the content of A to B, placing the result in C. These are the steps to execute this instruction:

1. Fetch and decode the instruction (ADD).
2. Fetch A.
3. Fetch B.
4. Add A and B.
5. Store the sum in C.

If we fault when we try to store in C page not currently in memory, we will have to get the desired page, bring it in, correct the page table, and restart the instruction. The restart will fetch the instruction again, decoding it again, fetching the two operands again, and then adding again. However, there is not much repeated work (less than one complete instruction), and the repetition is necessary only when a page fault occurs.

The major difficulty arises when one instruction may modify several different locations. This problem can be solved in two different ways. In one solution, the microcode computes and attempts to access both ends of both blocks. If a page fault is going to occur, it will happen at this step, before anything is modified. The move can then take place; we know that no page fault can occur, since all the relevant pages are in memory. The other solution uses temporary registers to hold the values of overwritten locations. If there is a page fault, all the old values are written back into memory before the trap occurs. This action restores the memory to its state before the instruction was started, so that the instruction can be repeated.

PERFORMANCE OF DEMAND PAGING

Demand paging can significantly affect the performance of a computer system. To see why, let's compute the **Effective Access Time** for a demand-paged memory. For most computer systems, the memory-access time, denoted *ma,*ranges from 10 to 200 nanoseconds. As long as we have no page faults, the effective access time is equal to the memory access time. If, a page fault occurs, we must first read the relevant page from disk and then access the desired word. Let p be the probability of a page fault ($0 \leq p \leq 1$). We would expect p to be close

to zero—that is, we would expect to have only a few page faults. The **Effective Access Time** is calculated as,

Effective Access Time = $(1 - p) \times ma + p \times$ page fault time.

TRANSLATION LOOK-ASIDE BUFFER

Operating system has its own methods for storing page tables and allocating a page table for each process. A pointer to the page table is stored with the other register values (like the instruction counter) in the process control block. When the dispatcher is told to start a process, it must reload the user registers and define the correct hardware page-table values from the stored user page table. 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**, with very high-speed logic to make the paging-address translation efficient. 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 page table 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 problem with this approach is the time required to access a user memory location. If we want to access location *i*, we must first index into the page table, using the value in the PTBR offset by the page number for *i*. This task requires a memory access. It provides us with the frame number, which is combined with the page offset to produce the actual address, to access the desired place in memory.With this scheme, *two* memory accesses are needed to access a byte (one for the page-table entry, one for the byte). Thus, memory access is slowed by a factor of 2. This delay would be intolerable under most circumstances. We might as well resort to swapping! 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 value. 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.

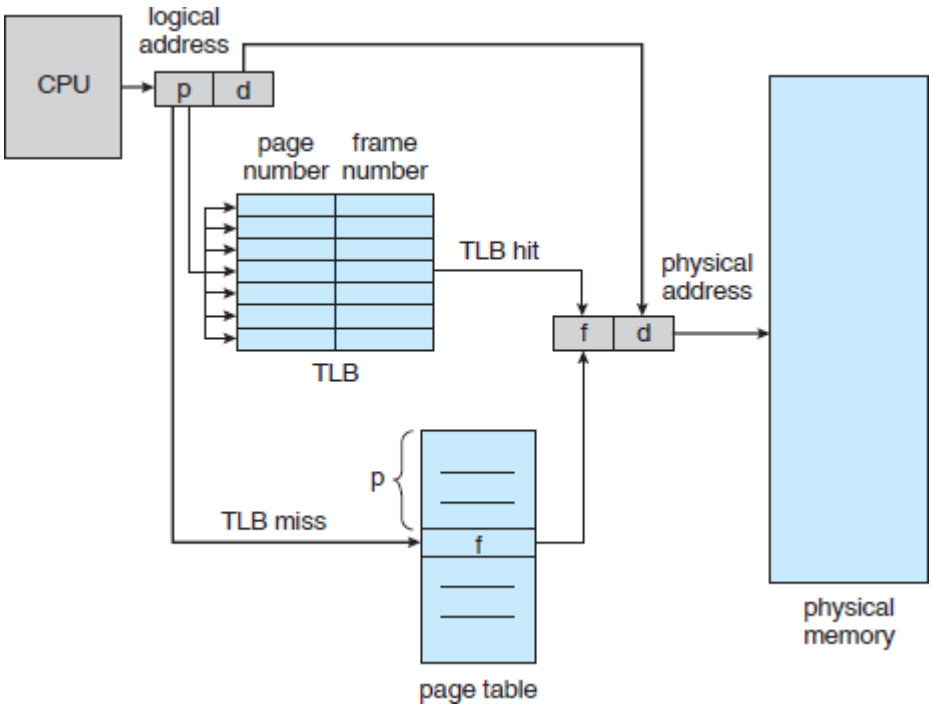


Figure 17: Paging Hardware with TLB.

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, 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 as shown in figure 17. In addition, we add the page number and frame number to the TLB, so that the next reference can be found quickly. 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. Some TLBs store **address-space identifiers (ASIDs)** in each TLB entry.

An ASID uniquely identifies each process and is used to provide address-space protection for that process. When the TLB attempts to resolve virtual page numbers, it ensures that the ASID for the currently running process matches the ASID associated with the virtual page. If the ASIDs do not match, the attempt is treated as a TLB miss. In addition to providing address-space protection, an ASID allows the TLB to contain entries for several different

processes simultaneously. If the TLB does not support separate ASIDs, then every time a new page table is selected with each context switch, and the TLB must be **flushed** (or erased) to ensure that the next executing process does not use the wrong translation information. Otherwise, the TLB could include old entries that contain valid virtual addresses but have incorrect or invalid physical addresses left over from the previous process. The percentage of times that a particular page number is found in the TLB is called the **hit ratio.**

An 80-percent hit ratio, for example, 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 the case by its probability:

$$\text{Effective Access Time} = 0.80 \times 120 + 0.20 \times 220 = 140 \text{ nanoseconds.}$$

In this example, we suffer a 40-percent slowdown in memory-access time.

4. 10 INVERTED PAGE TABLES

The purpose of this form of page management is to reduce the amount of physical memory needed to track virtual-to-physical address translations. We accomplish this savings by creating a table that has one entry per page of physical memory, indexed by the pair,

<process-id, page-number>

the information about which virtual memory page is stored in each physical frame, reduces the amount of physical memory needed to store the information in the inverted page tables. Also, the inverted page table no longer contains complete information about the logical address space of a process, and the information required if a referenced page is not currently in memory. Demand paging requires this information to process page faults. For the information to be available, an external page table (one per process) must be maintained. Each such table looks like the traditional per-process page table and contains information on where each virtual page is located. Since these tables are referenced only when a page fault occurs, they do not need to

be available quickly. Instead, they are themselves paged in and out of memory as necessary. Unfortunately, a page fault may now cause the virtual memory
manager to generate another page fault as it pages is in the external page table and it needs to locate the virtual page on the backing store. This special case requires careful handling in the kernel and a delay in the page-lookup processing.

PAGE REPLACEMENT

Page replacement takes the following approach. If no frame is free, we find one that is not currently being used and free it. We can free a frame by writing its contents to swap space and changing the page table and all other tables to indicate that the page is no longer in memory Figure 18.

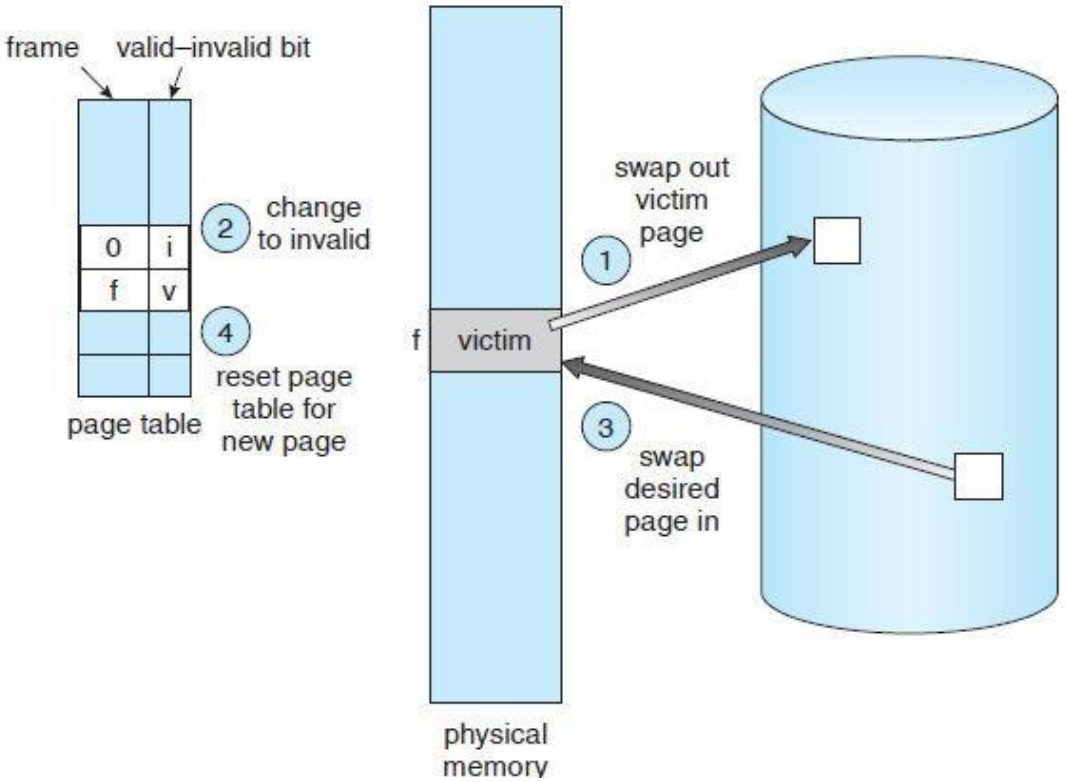


Figure 18: Page replacement.

We can now use the freed frame to hold the page for which the process faulted. We modify the page-fault service routine to include page replacement:

1. Find the location of the desired page on the disk.
2. Find a free frame:

    a. If there is a free frame, use it.

b. If there is no free frame, use a page-replacement algorithm to select a victim **frame**.

c. Write the victim frame to the disk; change the page and frame tables accordingly.

3. Read the desired page into the newly freed frame; change the page and frame tables.

4. Restart the user process.

Also, if no frames are free, *two* page transfers (one out and one in) are required. This situation effectively doubles the page-fault service time and increases the effective access time accordingly. This overhead can be reduced by using a **modify bit** (or **dirty bit**). The modify bit for a page is set by the hardware whenever any word or byte in the page has been modified. When we select a page for replacement, we examine its modify bit. If the bit is set, it means that the page has been modified since it was read in from the disk. In this case, we must write the page to the disk. If the modify bit is not set, however, the page has *not* been modified since it was read into memory. In this case, we need not write the memory page to the disk: it is already there. This technique also applies to read-only pages and such pages cannot be modified; they may be discarded when desired. This scheme can significantly reduce the time required to service a page fault, since it reduces I/O time by one-half *if* the page has not been modified.

Page replacement is basic to demand paging. It completes the separation between logical memory and physical memory. If a page that has been modified and it is to be replaced, its contents are copied to the disk, after replacement the reference to that page will cause a page fault. At that time, the page will be brought back into memory, perhaps replacing some other page in the process. perhaps replacing some other page in the process.

To solve this two major problems to implement demand paging, we must develop a **frame-allocation algorithm** and a **page-replacement algorithm**. That is, if we have multiple processes in memory, we must decide how many frames to allocate to each process; when page replacement is required, and also select the frames that are to be replaced. There are many different page-replacement algorithms. Every operating system probably has its own replacement scheme.

In general, if we want to replace a page, we must select the one with the lowest page-fault rate. We evaluate an algorithm by running it on a particular string of memory references and computing the number of page faults. The string of memory references is called a

reference string. Several page-replacement algorithms are illustrated with the following reference string

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

for a memory with three frames.

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. 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 19. Every time a fault occurs, we show which pages are in our three frames. There are fifteen faults altogether.

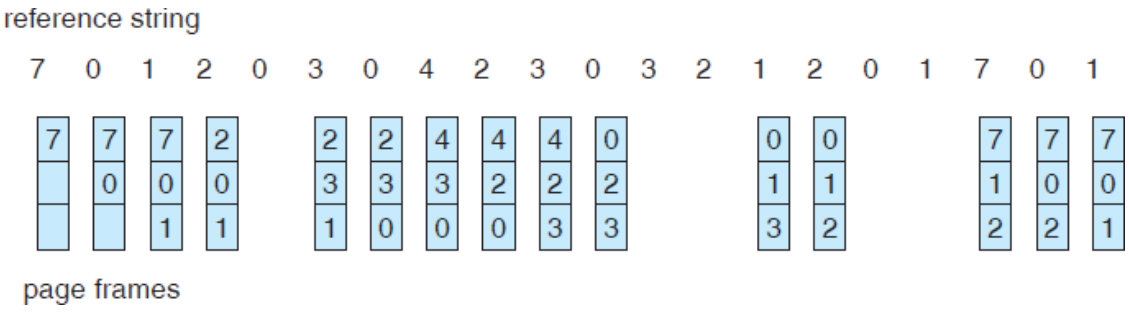


Figure 19: FIFO page-replacement algorithm

The FIFO page-replacement algorithm is easy to understand and program. However, its performance is not always good. 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. Thus, a bad replacement choice increases the page-fault rate and slows process execution. To illustrate the problems that are possible with a FIFO page-replacement algorithm, we consider the following reference string:

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

which results as Figure 20, with the number of faults for four frames as ten that is *greater* than the number of faults for three frames which is nine. This unexpected result is known as **Belady's anomaly**. (i.e) the page-fault rate may *increase* as the number of allocated frames increases.

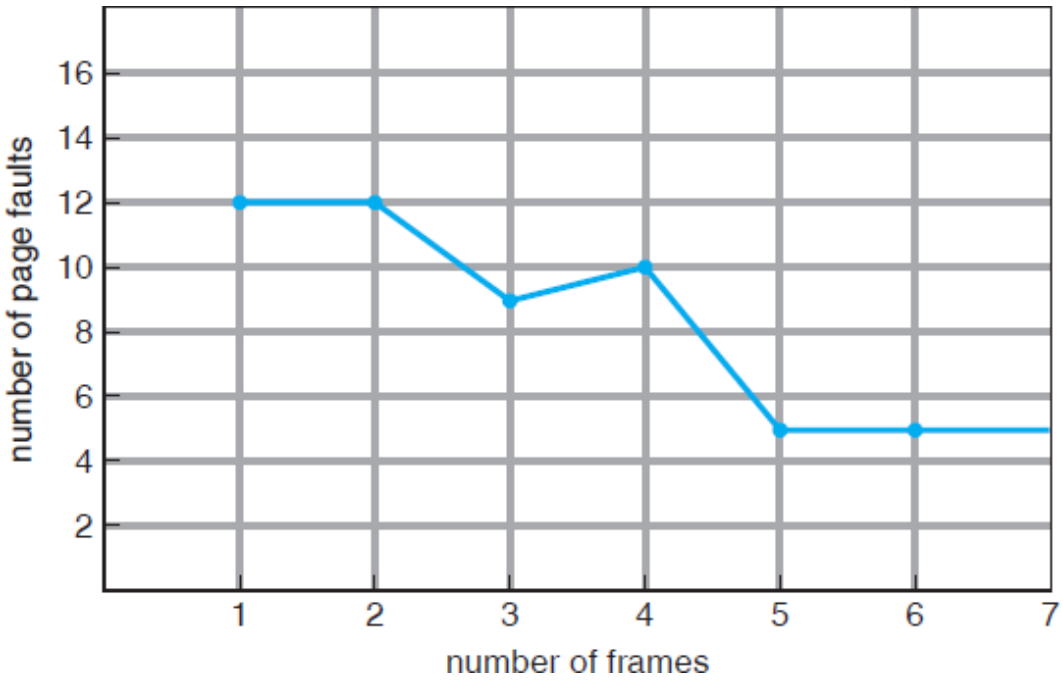


Figure 20: Page-fault curve for FIFO replacement on a reference string

OPTIMAL PAGE REPLACEMENT

An **optimal page-replacement algorithm**, which has the lowest page-fault rate of all algorithms and will never suffer from Belady's anomaly. This algorithm is called as OPT or MIN. The policy of Optimal Page Replacement is.

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 21. The first three references cause faults that fill the three empty frames. The reference to page 2 replaces page 7, because page 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 results in fifteen faults. Unfortunately, the optimal page-replacement algorithm is difficult to implement, because it requires future knowledge of the reference string.

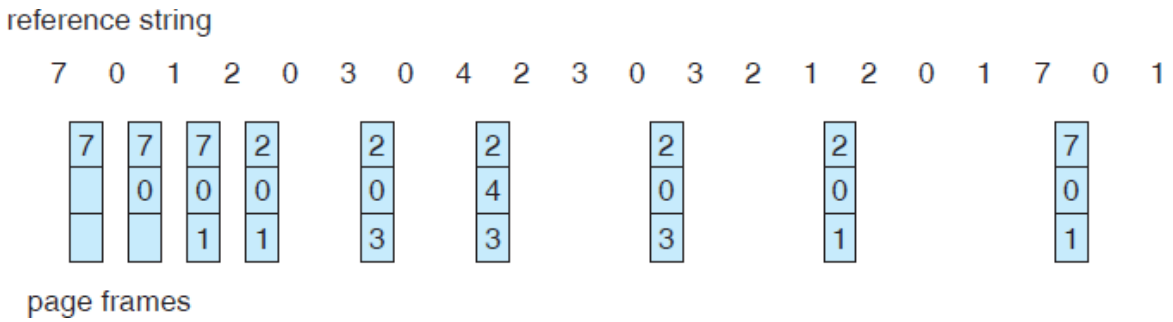


Figure 21: Optimal Page-Replacement Algorithm

LRU PAGE REPLACEMENT

If the optimal algorithm is not feasible, an approximation of the optimal algorithm is possible. The difference between the FIFO and OPT algorithms 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. 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. The result of applying LRU replacement to our example reference string is shown in Figure 22.

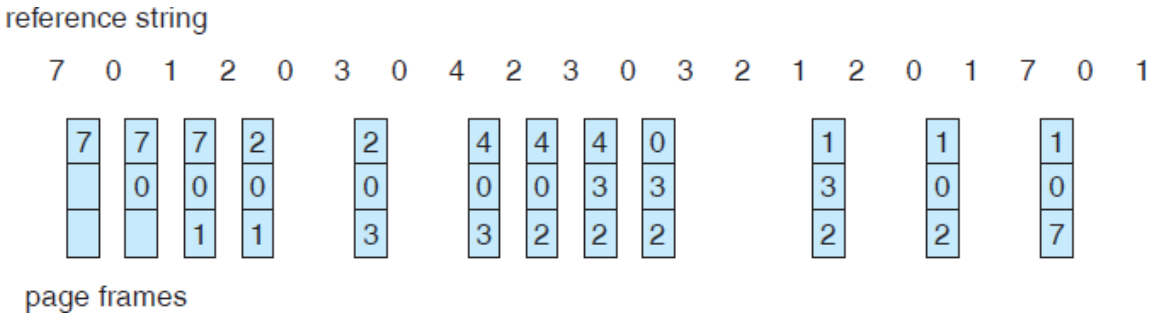


Figure 22: LRU page-replacement algorithm

The LRU algorithm produces twelve page faults. The first five faults are the same as those for optimal replacement. When the reference to page 4 occurs, the LRU replacement sees that, out of the three frames in memory, page 2 was used least recently. Thus, the LRU algorithm replaces page 2, without knowing that page 2 is about to be used. When faults for page 2 occurs 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 twelve faults is much better than FIFO replacement with fifteen page fault. 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 hardware assistance. Two implementations are feasible for LRU page-replacement:

(i) **COUNTERS:** 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. 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 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. Here, overflow of the clock must be considered.

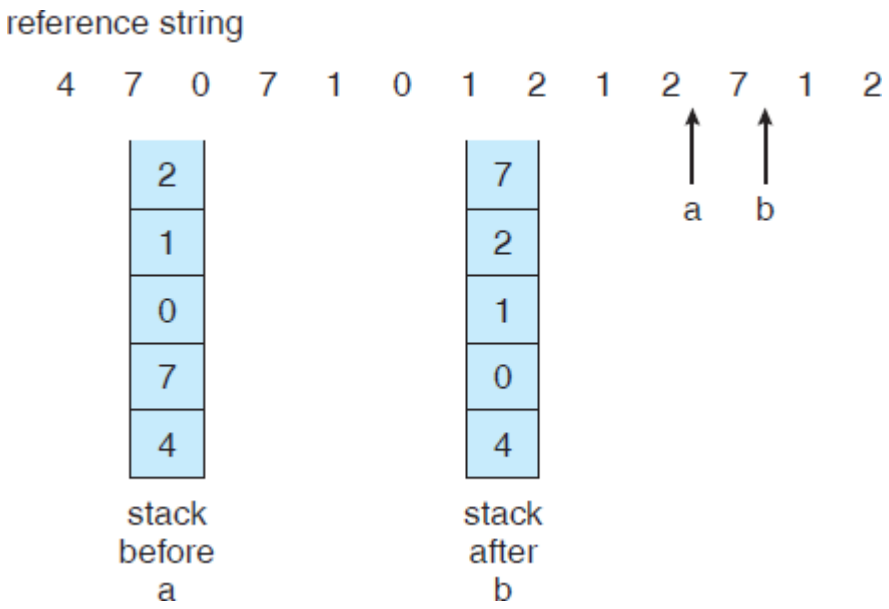


Figure 22: Use of a stack to record the most recent page references

(ii) **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 as shown in 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 pointer and a tail pointer. Removing a page and putting it on the top of the stack then requires changing six pointers at worst, 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.

LRU-APPROXIMATION PAGE REPLACEMENT

The basis for many page-replacement algorithms that approximate LRU replacement is to execute a user process where, the bit associated with each page referenced is set (to 1) by the hardware. After some time, we can determine which pages have been used and which have not been used by examining the reference bits, although we do not know the *order* of use.

ADDITIONAL-REFERENCE-BITS ALGORITHM

The additional ordering information by recording the reference bits at regular intervals can maintain an 8-bit byte for each page in a table in memory. At regular intervals the operating system shifts the reference bit for each page into the high-order bit of its 8-bit byte, shifting the other bits right by 1 bit and discarding the low-order bit. These 8-bit shift registers contain the history of page use for the last eight time periods. If the shift register contains 00000000, then the page has not been used for eight time periods. And a page that is used at least once in each period will have a shift register value of 11111111. If we interpret these 8-bit bytes as unsigned integers, the page with the lowest number is the LRU page can be replaced. We can replace by swapping out all pages with the smallest value by using FIFO method to choose among them. In the extreme case, the number can be reduced to zero, leaving only the reference bit itself. This algorithm is called the second-chance page-replacement algorithm.

SECOND-CHANCE ALGORITHM

The basic algorithm of second-chance replacement is a FIFO replacement algorithm. When a page has been selected, however, we inspect its reference bit. If the value is 0, we proceed to replace this page; but if the reference bit is set to 1, we give the page a second chance and move on to select the next FIFO page. When a page gets a second chance, its reference bit is cleared, and its arrival time is reset to the current time. Thus, a page that is given a second chance will not be replaced until all other pages have been replaced or given a second chances. One way to implement the second-chance algorithm is to have a reference to the *clock* algorithm as a circular queue. A pointer that is on the clock will indicate which page is to be replaced next. When a frame is needed, the pointer advances until it finds a page with a 0 reference bit. As it advances, it clears the reference bits as in Figure 23. Once a victim page is found, the page is replaced, and the new page is inserted in the circular queue in that

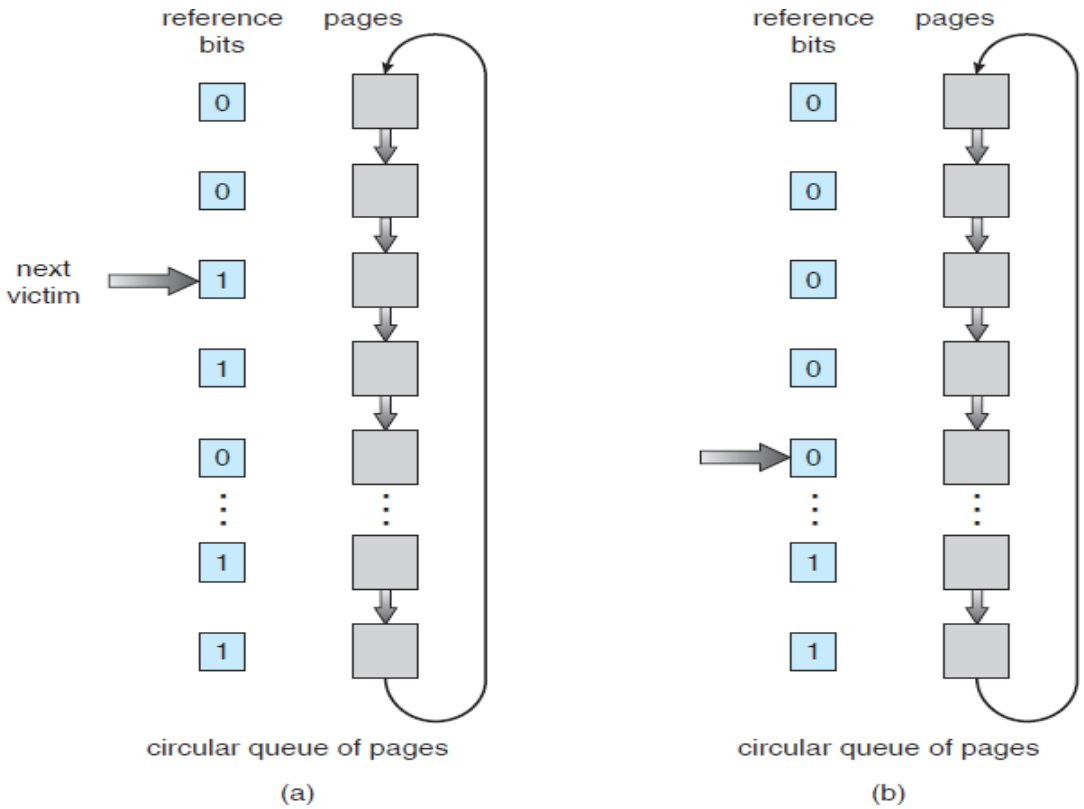


Figure 23: Second-Chance (Clock) Page-Replacement Algorithm

position. The worst case is when all bits are set in the pointer cycles, the whole queue gives each page a second chance. It clears all the reference bits before selecting the next page for replacement. Second-chance replacement degenerates to FIFO replacement if all bits are set.

ENHANCED SECOND-CHANCE ALGORITHM

Enhanced second-chance algorithm considers the reference bit and the modify bit in an ordered pair. With these two bits, we have the following four possible classes:

1. (0, 0) neither recently used nor modified, it is best page to replace.
2. (0, 1) not recently used but modified, it is not good to replace because the page is needed to be written out before replacement.
3. (1, 0) recently used but clean, it can probably be used again soon.
4. (1, 1) recently used and modified, it will probably be used again soon, and the page will be need to be written out to disk before it can be replaced.

Each page is in one of these four classes. When page replacement is called, the same scheme as in the clock algorithm; will examine whether the page to which we are pointing has the reference bit set to 1 and scan the circular queue several times before we find a page to be replaced. The major difference between this algorithm and the simpler clock algorithm is that here we give preference to those pages that have been modified to reduce the number of I/Os required.

COUNTING-BASED PAGE REPLACEMENT

There are many other algorithms that can be used for page replacement. For example, we can keep a counter of the number of references that have been made to each page and develop the following two schemes.

(i) The **Least-Frequently-Used** (**LFU) page-replacement algorithm** requires that the page with the smallest count be replaced. The reason for this selection is that an actively used page should have a large reference count. A problem arises, however, when a page is used heavily during the initial phase of a process but then is never used again. Since it was used heavily, it has a large count and remains in memory even though it is no longer needed. One solution is to shift the counts right by 1 bit at regular intervals, forming an exponentially decaying average usage count.

(ii) The **Most-Frequently-Used** (**MFU) page-replacement algorithm** is based on the argument that the page with the smallest count is just brought in and has yet to be used. As you expect, neither MFU nor LFU replacement is common. The implementation of these algorithms is expensive, and they do not approximate OPT replacement well.

Questions and Answers:

2 Marks:

1. Differentiate internal and external fragmentation.

   Internal fragmentation: Memory that is internal to a partition but not being used

   External fragmentation: Total memory space exists to satisfy a request, but it is not contiguous.

2. What is meant by Paging? Give its advantages.

   Paging is a Memory-management scheme that permits the physical -address space of a process to be Non-contiguous.

   Advantages: (i) Avoids the considerable problem of fitting the varying-sized memory chunks onto the backing store. (ii) Fragmentation problems are also prevalent backing store, except that access is much slower, so compaction is impossible.

3. What is meant by Locality of reference?

   During any phase of execution, the page references only a relative small fraction of its pages. This reference of fraction of all pages is called as Locality of Reference.

4. Differentiate Segmentation and Paging storage.

S. No.	Segmentation	Paging
1.	The physical memory is breaking into variable-sized blocks called segments.	The physical memory is breaking into fixed-sized blocks called frames and logical memory is breaking into blocks of the same size called pages.
2.	Address generated by CPU is divided into segment number (S)	Address generated by CPU is divided as Page number (p) and Page offset (d).

	and segment offset (d).	
3.	Physical address = segment base + offset	Physical address = page size * frame number + offset
4.	Has external fragmentation	No external fragmentation

5. What is meant by Page Fault?

   Whenever memory management unit accessing the page that are not in the memory is called as Page Fault.

6. What is meant by Swapping?

   It is a process of bringing in each process in its entirety, running it for a while, then putting it back on the disk.

7. What is meant by Memory Compaction?

   When swapping creates multiple holes in memory, it is possible to combine them all into one big by moving all the processes downward as far as possible.

8. What is demand paging?

   Swapping a page into the memory, when we want to execute a process. Also called as Lazy swapper because the page is brought into the memory on demand

9. Define the virtual memory? What are its advantages?

   Virtual memory is a technique that allows the execution of processes that are not completely in memory.

   Advantages:

   - Enables users to run programs that are larger than actual physical memory.
   - VM makes the task of programming much easier.
   - Virtual memory allows processes to share files easily and to implement shared memory.
   - It provides an efficient mechanism for process creation.

10. How can measure the performance of demand paging?

    To measure the demand paging , the **effective access time** for a demand –paged memory is calculated by:

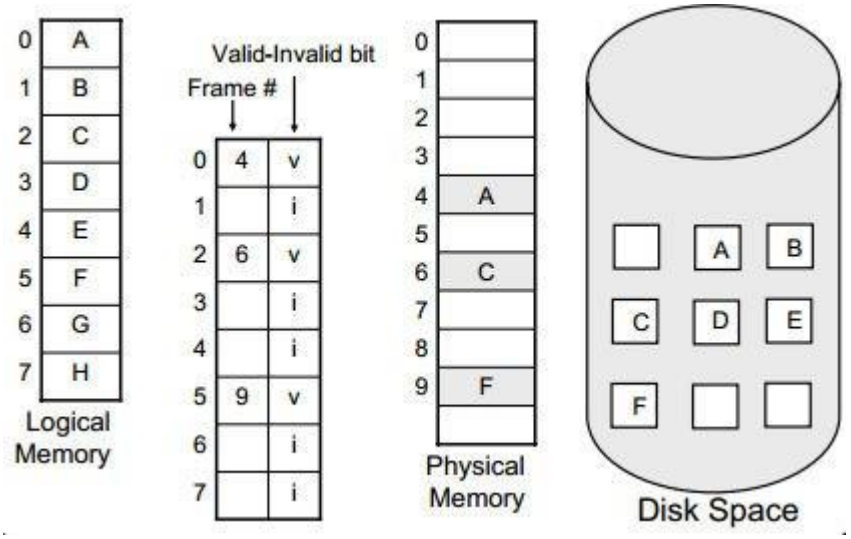
    Effective access time = (1 – p) x ma + p x page fault time

    Where, **p**: The probability of page fault, $0 < p < 1$;

    ma: Memory access time , ranges from 10 to 200 nanosecond.

11. How can the system distinguish between the pages that are in main memory from the pages that are on the disk?

    The system uses valid-invalid bit is used. This bit is set to "valid" when the page in memory, while it set to "invalid" when the page either not valid or is the page is valid but is on the disk, as in the following figure.



12. What are the differences between pager and swapper?

S. No.	Pager	Swapper
1.	Pager Swaps a page into memory when this page will be needed into memory.	Swapper swaps the entire processes into memory
2.	It use in demand-paging system	It uses in paging system

Answer in detail

1. Discuss in detail paging.

   Hints: Paging definition, Basic method-page, frame, page table, page offset and page number, Paging hardware diagram, TLB with diagram, Protection bits and valid/invalid bits.

2. Bring out a detailed study on Segmentation.

   Hints: User view of program; Segmentation definition; Hardware - with diagram; Protection and sharing with diagram; Fragmentation

3. Discuss in brief about Demand paging.

   Hints: Definition: Lazy Swapper, Explanation : Page Fault, Page Fault Trap, Example, Effective Access Time

4. What are the steps to modify the page-fault service routine to include page replacement?

   Step 1. Find the location of the desired page on the disk.

   Step 2. Find a free frame:

   a) If there is a free frame, use it.

   b) If there are no free frames, use a page-replacement algorithm to select a **victim frame.**

   c) Write the victim frame to the disk, change the pages table.

   Step 3. Read the desired page and store it in the free frame. Adjust the page table.

   Step 4. Restart the user process.

5. Explain in detail the various page replacement strategies.

   Hints: Page replacement basic scheme with diagram; FIFO page replacement; optimal page replacement; LRU page replacement; LRU approximation page replacement; Counting-based page replacement; Page buffering algorithm.

PROBLEMS:

1. Calculate the size of memory if its address consists of 22 bits and the memory is 2-byte addressable.

Solution:

Given,

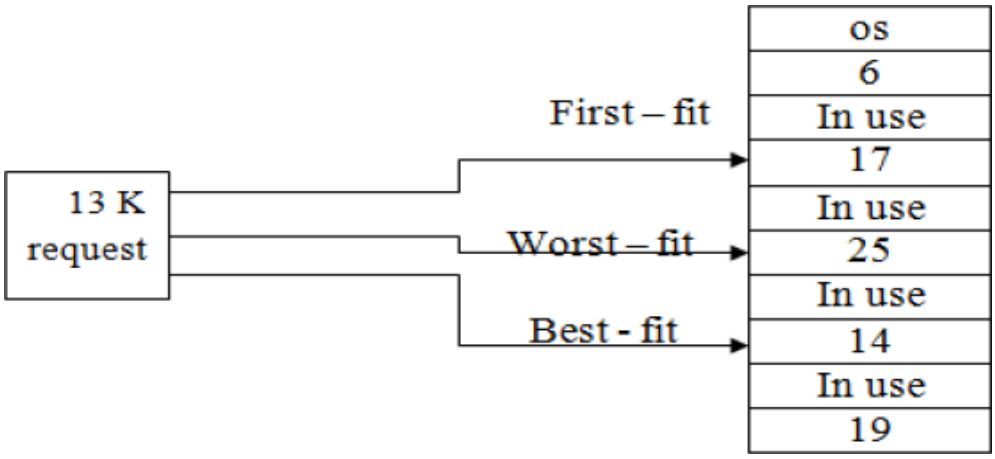
- Number of locations possible with 22 bits = 2^{22} locations
- It is given that the size of one location = 2 bytes

Formula: Size of memory = 2^n x Size of one location.

Thus, Size of memory = 2^{22} x 2 bytes= 2^{23} bytes = 8 MB

2. Suppose that we have free segments with sizes: 6, 17, 25, 14, and 19. Place a program with size 13KB in the free segment using first-fit, best-fit and worst fit?

Solution:



2. Consider a system with byte-addressable memory, 32 bit logical addresses, 4 kilobyte page size and page table entries of 4 bytes each. The size of the page table in the system in megabytes is .

Solution:

Formula:

(i) Size of page table = Number of entries in page table x Page table entry size

(ii) Number of entries in pages table = Number of pages the process is divided

(iii) Page table entry size = Number of bits in frame number + Number of bits used for optional fields, if any

Given ,

- Number of bits in logical address = 32 bits
- Page size = 4KB
- Page table entry size = 4 bytes

Process Size:

Number of bits in logical address = 32 bits

Thus, Process size = 2^{32} B = 4 GB

Number of Entries in Page Table:

Number of pages the process is divided = Process size / Page size = 4 GB / 4 KB = 2^{20} pages

Thus, Number of entries in page table = 2^{20} entries

Page Table Size:

Page table size = Number of entries in page table x Page table entry size

= 2^{20} x 4 bytes = 4 MB

3. Assume an average page-fault service time is 25 milliseconds and a memory access time is 100 nanoseconds. Find the Effective Access Time?

Effective Access Time (EAT)= (1 – p) x (ma) + p x (page fault time)

= (1 – p) x 100 + p x 25,000,000

= 100 – 100 x p + 25,000,000 x p

4. Consider a program consists of 5 segments: S0 = 600, S1 = 14 KB, S2= 100 KB, S3 =580 KB, and S4 = 96 KB. Assume at that time, the available free space partitions of memory are 1200–1805, 50 – 150, 220-234, and 2500-3180.

Find the following:

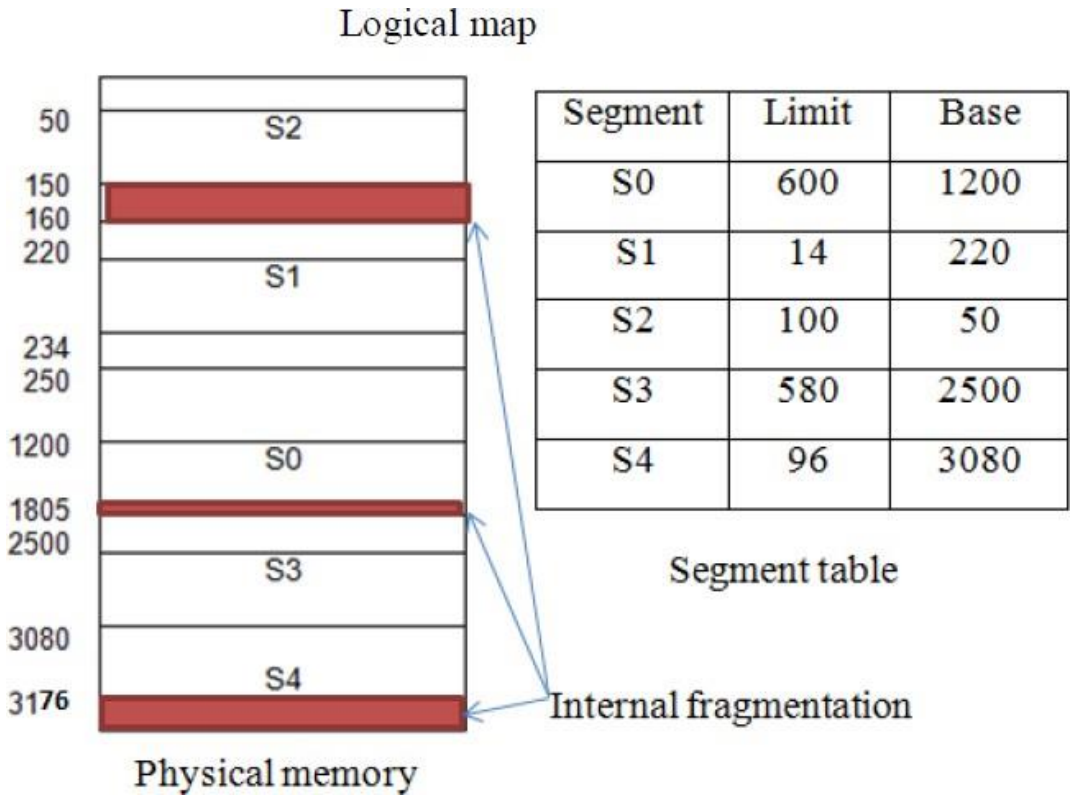
a. Allocate space for each segment in memory?

b. Calculate the external fragmentation and the internal fragmentation?

c. What are the addresses in physical memory for the following logical addresses:

(i) 0.580, (ii) 1.17 (iii) 2.66 (d) 3.82 (iv) 4.20?

Solution for a:



Segment	Limit	Base
S0	600	1200
S1	14	220
S2	100	50
S3	580	2500
S4	96	3080

Segment table

Solution for b:

External Fragmentation =0.

Internal Fragmentation = (160-150) +(1805-1800) + (3180-3176)

= 10 + 5 + 4 = 19

Fragmentation = External Fragmentation + Internal Fragmentation = 0 + 19 = 19

Solution for c:

The physical addresses are

(i) 0.580------the physical address of 0.580 = 1200+580 = 1780.

(ii) 1.17 ----- Since d > limit of S1, the address is **wrong**.

(iii) 2.66 ----- the physical address of 2.66= 50 + 66 = 116

(iv) 3.82 ----- the physical address is of 3.82 is = 2500 + 82 = 2582

(v) 4.20 ----- the physical address 4.20 = 3080+20 = 3100

5. Consider the following page reference using **three** frames that are initially empty. Find the page faults using FIFO algorithm, where the page reference sequence: 7,0,1, 2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1?

Solution:

7	**0**	**1**	**2**	**0**	**3**	**0**	**4**	**2**	**3**	**0**	**3**	**2**	**1**	**2**	**0**	**1**	**7**	**0**	**1**
7	7	7	0	0	1	2	3	0	4	2	2	2	3	0	0	0	1	2	7
	0	0	1	1	2	3	0	4	2	3	3	3	0	1	1	1	2	7	0
		1	2	2	3	0	4	2	3	0	0	0	1	2	2	2	7	0	2
*	*	*	*		*	*	*	*	*	*			*	*			*	*	*

Page fault

The page fault = 15.

6. Let the page fault service time be 10 ms in a computer with average memory access time being 20 ns. If one page fault is generated for every 10^6 memory accesses, what is the Effective Access Time for the memory?

Solutions:

Given-

- Page fault service time = 10 ms
- Average memory access time = 20 ns
- One page fault occurs for every 10^6 memory accesses

Page Fault Rate:

- It is given that one page fault occurs for every 10^6 memory accesses.

Thus, Page fault rate = $1 / 10^6 = 10^{-6}$

Effective Access Time (EAT) with Page Fault:

It is given that effective memory access time without page fault = 20 ns.

Now, substituting values in the above formula,

we get,

EAT with page fault = 10^{-6} x { 20 ns + 10 ms } + (1 – 10^{-6}) x { 20 ns }

= 10^{-6} x 10 ms + 20 ns

= 10^{-5} ms + 20 ns

= 10 ns + 20 ns

= 30 ns

7. Consider a system with a two-level paging scheme in which a regular memory access takes 150 nanoseconds and servicing a page fault takes 8 milliseconds. An average instruction takes 100 nanoseconds of CPU time and two memory accesses. The TLB hit ratio is 90% and the page fault rate is one in every 10,000 instructions. What is the effective average instruction execution time?

Solutions:

Given,

- Number of levels of page table = 2
- Main memory access time = 150 ns
- Page fault service time = 8 msec
- Average instruction takes 100 ns of CPU time and 2 memory accesses
- TLB Hit ratio = 90% = 0.9
- Page fault rate = $1 / 10^4 = 10^{-4}$

Assume TLB access time = 0 since it is not given in the question.

Also, TLB access time is much less as compared to the memory access time.

Effective Access Time without Page Fault:

Substituting values in the above formula, we get

Effective memory access time without page fault

$= 0.9 \times \{ 0 + 150 \text{ ns} \} + 0.1 \times \{ 0 + (2+1) \times 150 \text{ ns} \}$

$= \{ 0.9 \times 150 \text{ ns} \} + \{ 0.1 \times 450 \text{ ns} \}$

$= 135 \text{ ns} + 45 \text{ ns}$

$= 180 \text{ ns}$

Effective Access Time with Page Fault:

Substituting values in the above formula, we get-

Effective access time with page fault

$= 10^{-4} \times \{ 180 \text{ ns} + 8 \text{ msec} \} + (1 - 10^{-4}) \times 180 \text{ ns}$

$= 8 \times 10^{-4} \text{ msec} + 180 \text{ ns}$

$= 8 \times 10^{-7} \text{ sec} + 180 \text{ ns}$

$= 800 \text{ ns} + 180 \text{ ns}$

= 980 ns

Effective Average Instruction Execution Time:

Effective Average Instruction Execution Time

= 100 ns + 2 x Effective memory access time with page fault

= 100 ns + 2 x 980 ns

= 100 ns + 1960 ns

= 2060 ns

8. A demand paging system takes 100 time units to service a page fault and 300 time units to replace a dirty page. Memory access time is 1 time unit. The probability of a page fault is p. In case of a page fault, the probability of page being dirty is also p. If, it is observed that the average access time is 3 time units. What is the value of p?

Solution:

Given,

- Page fault service time = 100 time units
- Time taken to replace dirty page = 300 time units
- Average memory access time = 1 time unit
- Page fault rate = p
- Probability of page being dirty = p
- Effective access time = 3 time units

Now, According to question,

3 time units = p x { 1 time unit + p x { 300 time units } + (1 – p) x { 100 time units } } + (1 – p) x { 1 time unit }

$3 = p \times \{ 1 + 300p + 100 – 100p \} + (1 – p)$

$3 = p \times \{ 101 + 200p \} + (1 – p)$

$3 = 101p + 200p^2 + 1 – p$

$3 = 100p + 200p^2 + 1$

$200p^2 + 100p – 2 = 0$

On solving this quadratic equation, we get p = 0.019258