R.M.K GROUP OF ENGINEERING INSTITUTIONS

R.M.K
GROUP OF
INSTITUTIONS

R.M.K GROUP OF INSTITUTIONS

R.M.K
GROUP OF
INSTITUTIONS

R.M.K GROUP OF INSTITUTIONS

Please read this disclaimer before proceeding:

This document is confidential and intended solely for the educational purpose of RMK Group of Educational Institutions. If you have received this document through email in error, please notify the system manager. This document contains proprietary information and is intended only to the respective group / learning community as intended. If you are not the addressee you should not disseminate, distribute or copy through e-mail. Please notify the sender immediately by e-mail if you have received this document by mistake and delete this document from your system. If you are not the intended recipient you are notified that disclosing, copying, distributing or taking any action in reliance on the contents of this information is strictly prohibited.

R.M.K GROUP OF INSTITUTIONS

22CS304
OPERATING SYSTEMS

UNIT I

Department : **AI & DS**

Batch/Year : **2022 - 2026 /IV**

Created by : **Dr.B.PrathushaLaxmi**
Ms.A.Akila

Date : **11.01.2024**

R.M.K GROUP OF INSTITUTIONS

Table of Contents

R.M.K GROUP OF INSTITUTIONS

Course Objectives

22CS4304 OPERATING SYSTEMS

COURSE OBJECTIVES

- To explain the basic concepts of operating systems and process.
- To discuss threads and implement various CPU scheduling algorithms.
- To describe the concept of process synchronization and implement deadlocks
- To analyse various page replacement schemes.
- To investigate disk scheduling algorithms

Prerequisite

22CS304 OPERATING SYSTEMS

PREREQUISITE

COMPUTER ORGANIZATION AND ARCHITECTURE

R.M.K GROUP OF INSTITUTIONS

Syllabus

R.M.K
GROUP OF
INSTITUTIONS

22CS304 - OPERATING SYSTEMS

SYLLABUS **3 0 0 3**

UNIT I INTRODUCTION TO OPERATING SYSTEMS AND PROCESSES

Introduction to OS –Computer system organization - architecture – Resource management - Protection and Security – Virtualization - Operating System Structures - Services - User and Operating-System Interface - System Calls - System Services - Design and Implementation - Building and Booting an Operating System - Process Concept - Process Scheduling - Operations on Processes – Inter process Communication - IPC in Shared-Memory Systems - IPC in Message-Passing Systems

UNIT II THREADS AND CPU SCHEDULING

Threads & Concurrency: Overview - Multicore Programming - Multithreading Models - Thread Libraries - Implicit Threading - Threading Issues - CPU Scheduling: Basic Concepts - Scheduling Criteria - Scheduling Algorithms - Thread Scheduling - Multi-Processor Scheduling - Real-Time CPU Scheduling

UNIT III PROCESS SYNCHRONISATION AND DEADLOCKS

Process Synchronization - The critical-section problem, Peterson's Solution - Synchronization hardware, Mutex locks, Semaphores, monitors, Liveness - Classic problems of synchronization – Bounded Buffer Problem - Reader's & Writer Problem, Dinning Philosopher Problem, Barber's shop problem. Deadlock - System model - Deadlock characterization, Methods for handling deadlocks - Deadlock prevention - Deadlock avoidance - Deadlock detection - Recovery from deadlock

UNIT IV MEMORY MANAGEMENT

Memory Management: Contiguous Memory Allocation - Paging - Structure of the Page Table – Swapping - Virtual Memory: Demand Paging – Copy-on write – Page Replacement – Allocation of frames – Thrashing Memory – Compression

UNIT V FILE MANAGEMENT

File Management: File Concept – Access Methods – Directory Structure – Protection - Memory-Mapped File - Disk Management: Disk structure, Disk scheduling - FCFS, SSTF, SCAN, C-SCAN, Disk reliability, Disk formatting, Boot-block, Bad blocks - I/O Hardware: I/O devices, Device controllers, Direct Memory Access - Case Study-Linux.

Course Outcomes

COURSE OUTCOMES

- CO1: Implement the operating system concepts and process.
- CO2: Analyse various CPU scheduling algorithms and thread mechanism.
- CO3: Implement process synchronization and deadlock problems
- CO4: Design various page replacement techniques to given situation
- CO5: Implement various disk scheduling techniques

CO – PO/ PSO Mapping

R.M.K
GROUP OF
INSTITUTIONS

CO-PO MAPPING

COs	PO's/PSO's														
	PO 1	PO 2	PO 3	PO 4	PO 5	PO 6	PO 7	PO 8	PO 9	PO 10	PO 11	PO 12	PSO 1	PSO 2	PSO 3
CO1	3	2	2	2	2	-	-	-	2	-	-	2	2	-	-
CO2	3	3	2	2	2	-	-	-	2	-	-	2	2	-	-
CO3	2	2	1	1	1	-	-	-	1	-	-	1	2	-	-
CO4	3	3	1	1	1	-	-	-	1	-	-	1	2	-	-
CO5	3	3	1	1	1	-	-	-	1	-	-	1	3	1	-

1 – Low, 2 – Medium, 3 – Strong

Lecture Plan

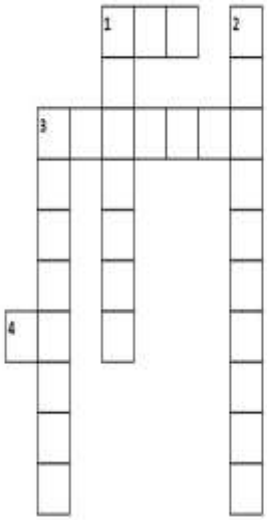
R.M.K
GROUP OF
INSTITUTIONS

LECTURE PLAN

S No	Topics	No of periods	Proposed date	Actual Lecture Date	Pertaining CO	Taxonomy level	Mode of delivery
1	Introduction to OS –Computer system organization	1			CO1	K1	ICT Tools
2	architecture – Resource management	1			CO1	K2	ICT Tools
3	Protection and Security – Virtualization	1			CO1	K2	ICT Tools
4	Operating System Structures - Services	1			CO1	K2	ICT Tools
5	User and Operating-System Interface - System Calls	1			CO1	K2	ICT Tools
6	System Services - Design and Implementation - Building and Booting an Operating System	1			CO1	K2	ICT Tools
7	Process Concept - Process Scheduling - Operations on Processes	1			CO1	K2	ICT Tools
8	Inter process Communication - IPC in Shared-Memory Systems - IPC in Message-Passing Systems	1			CO1	K2	ICT Tools
9	Revision	1			CO1	K2	ICT Tools

Activity Based Learning

1. Group discussion on VARIOUS OPERATING SYSTEMS
2.



Across

1. DEFINES A PROCESS TO OS

3. ACTIVE ENTITY

4. INTERFACE BETWEEN USER AND HARDWARE

Down

1. PASSIVE ENTITY

2. INTERFACE BETWEEN PROCESS AND OS

3. POWER OF 2

Lecture Notes

R.M.K
GROUP OF
INSTITUTIONS

1. Introduction:

Definition:

An operating system acts as an **intermediary between the user of a computer 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 **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.

Operating system goals:

- ❑ Execute user programs and make solving user problems easier
- ❑ Make the computer system convenient to use
- ❑ Use the computer hardware in an efficient manner

A computer system can be divided roughly into four components: ***the hardware, the operating system, the application programs, and a user*** (Figure 1.1). The **hardware**— the 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. The operating system controls the hardware and coordinates its use among the various application programs for the various users.

We can also view a computer system as consisting of hardware, software, and data. The **operating system** Controls and coordinates use of hardware among various applications and users. **Users** refer to People, machines, other computers

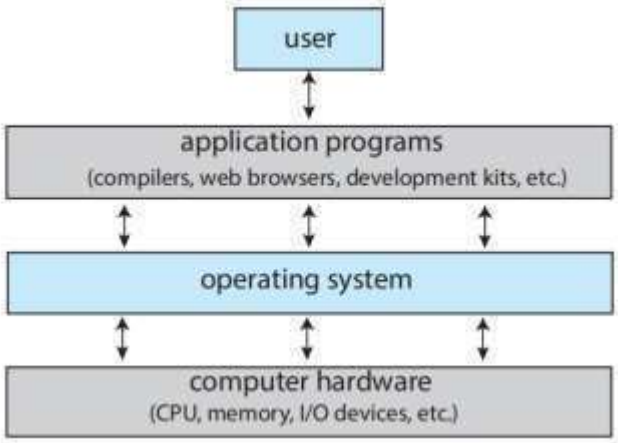


Figure 1.1 Abstract view of the components of a computer system.

2. 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 between components and shared memory (Figure 1.2).

Each **device controller** is in charge of a specific type of device (for example, a disk drive,audio device, or graphics display). Depending on the controller, more than one device may be attached. For instance, one system USB port can connect to a USB hub, to which several devices can connect. 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. 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.

1. Interrupts

Consider a typical computer operation: a program performing I/O. To start anvI/O operation, the device driver loads the appropriate registers in 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").

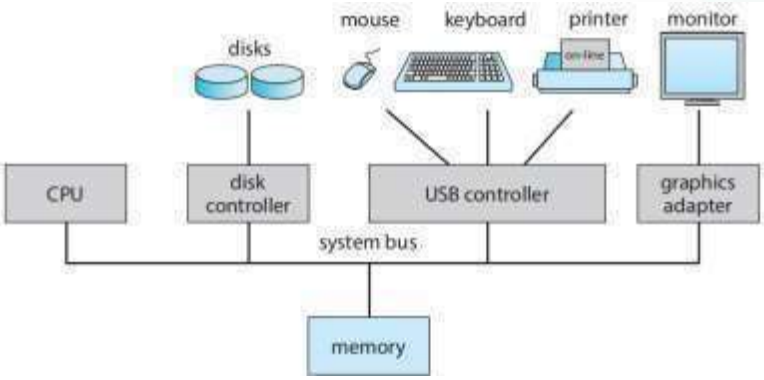


Figure 1.2 A typical PC computer system

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 that it has finished its operation. The device driver then gives control to other parts of 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 such as "write completed successfully" or "device busy". But how does the controller inform the device driver that it has finished its operation? This is accomplished via an **interrupt**.

1. Overview

- Hardware may trigger an interrupt at any time by sending a signal to the CPU, usually by way of the system bus. Interrupts are used for many other purposes as well and are a key part of how operating systems and hardware interact.
- 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. A timeline of this operation is shown in Figure 1.3.

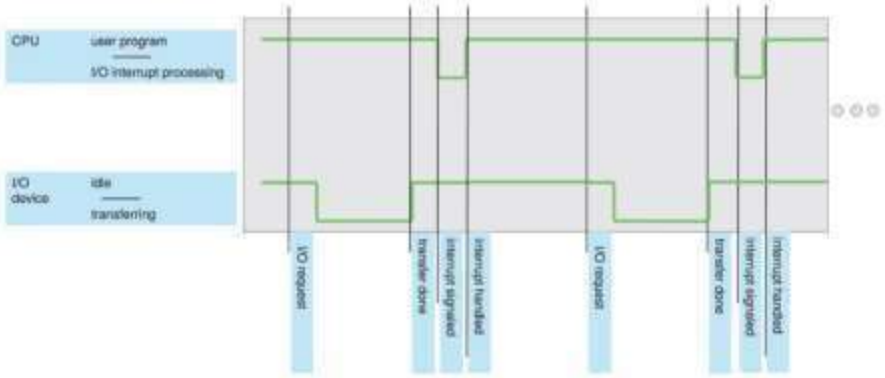


Figure 1.3 Interrupt timeline for a single program doing output.

- Interrupts are an important part of a computer architecture. Each computer design has its own interrupt mechanism, but several functions are common. **The interrupt must transfer control to the appropriate interrupt service routine.**
- The interrupt vector is the array of addresses of the interrupt service routines for the various devices.
- This i**nterrupt vecto**r is then indexed by a unique number, given with the interrupt request, to provide the address of the interrupt service routine for the interrupting device.
- The interrupt architecture must also save the state information of whatever was interrupted, so that it can restore this information after servicing the interrupt. If the interrupt routine needs to modify the processor state— for instance, by modifying register values—it must explicitly save the current state and then restore that state before returning.
- After the interrupt is serviced, the saved return address is loaded into the program counter, and the interrupted computation resumes as though the interrupt had not occurred.
- **1.2.1.2 Implementation**
- The basic interrupt mechanism works as follows. The CPU hardware has a wire called the **interrupt-request line** that the CPU senses after executing every instruction.
- When the CPU detects that a controller has asserted a signal on the interrupt-request line, it reads the interrupt number and jumps to the **interrupt-handler routine** by using that interrupt number as an index into the interrupt vector.
- It then starts execution at the address associated with that index. The interrupt handler saves any state it will be changing during its operation, determines the cause of the interrupt, performs the necessary processing, performs a state restore, and executes a return from interrupt instruction to return the CPU to the execution state prior to the interrupt.

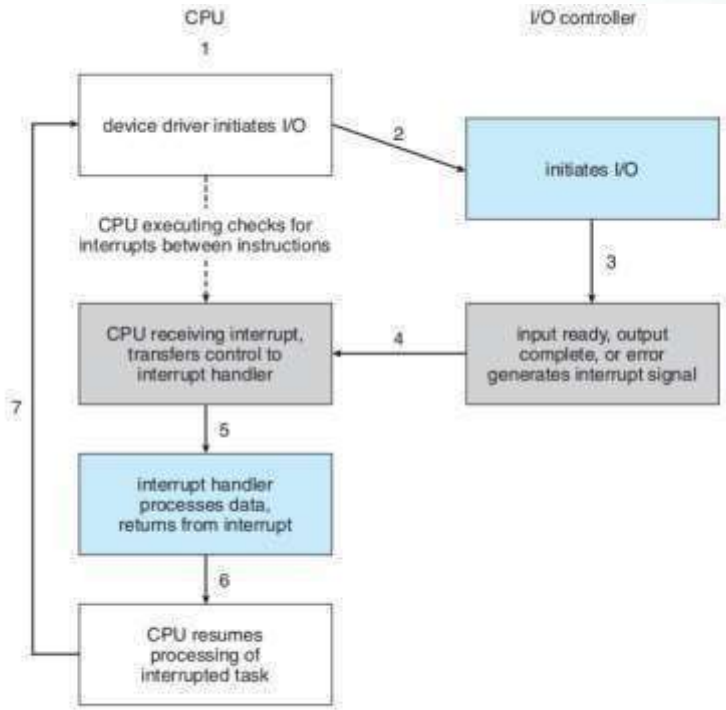


Figure 1.4 Interrupt-driven I/O cycle.

- The device controller **raises** an interrupt by asserting a signal on the interrupt request line, the CPU **catches** the interrupt and **dispatches** it to the interrupt handler, and the handler **clears** the interrupt by servicing the device.
- Figure 1.4 summarizes the interrupt-driven I/O cycle.
- In a modern operating system, however, there is the need for more sophisticated interrupt handling features.
- 1. To defer interrupt handling during critical processing.
- 2. Need an efficient way to dispatch to the proper interrupt handler for a device.
- 3. Need multilevel interrupts, so that the operating system can distinguish between high- and low-priority interrupts and can respond with the appropriate degree of urgency.
- In modern computer hardware, these three features are provided by the CPU and the interrupt-controller hardware. Most CPUs have two interrupt request lines. One is the **non-maskable interrupt,** which is reserved for events such as unrecoverable memory errors.

- The second interrupt line is **maskable**: it can be **turned off by the CPU before the execution of critical instruction sequences that must not be interrupted.** The maskable interrupt is used by device controllers to request service.
- In practice, however, computers have more devices (and, hence, interrupt handlers) than they have address elements in the interrupt vector. A common way to solve this problem is to use i**nterrupt chaining,** in which each element in the interrupt vector points to the head of a list of interrupt handlers.
- When an interrupt is raised, the handlers on the corresponding list are called one by one, until one is found that can service the request. This structure is a compromise between the overhead of a huge interrupt table and the inefficiency of dispatching to a single interrupt handler.
- Figure 1.5 illustrates the design of the interrupt vector for Intel processors. The events from 0 to 31, which are nonmaskable, are used to signal various error conditions. The events from 32 to 255, which are maskable, are used for purposes such as device-generated interrupts.
- The interrupt mechanism also implements a system of **interrupt priority levels.** These levels enable the CPU to defer the handling of low-priority interrupts without masking all interrupts and makes it possible for a high-priority interrupt to preempt the execution of a low-priority interrupt.

vector number	description
0	divide error
1	debug exception
2	null interrupt
3	breakpoint
4	INTO-detected overflow
5	bound range exception
6	invalid opcode
7	device not available
8	double fault
9	coprocessor segment overrun (reserved)
10	invalid task state segment
11	segment not present
12	stack fault
13	general protection
14	page fault
15	(Intel reserved, do not use)
16	floating-point error
17	alignment check
18	machine check
19–31	(Intel reserved, do not use)
32–255	maskable interrupts

Figure 1.5 Intel processor event-vector table.

Storage Structure

- The CPU can load instructions only from memory, so any programs must first be loaded into memory to run. General-purpose computers run most of their programs from rewritable memory, called main memory (also called random-access memory, or RAM).
- Main memory commonly is implemented in a semiconductor technology called dynamic random-access memory (DRAM). Computers use other forms of memory as well. For example, the first program to run on computer power-on is a bootstrap program, which then loads the operating system. Since RAM is volatile—loses its content when power is turned off or otherwise lost—we cannot trust it to hold the bootstrap program.
- The computer uses electrically erasable programmable read-only memory (EEPROM) and other forms of firmware —storage that is infrequently written to and is nonvolatile. EEPROM can be changed but cannot be changed frequently. In addition, it is low speed, and so it contains mostly static programs and data that aren't frequently used. For example, the iPhone uses EEPROM to store serial numbers and hardware information about the device.
- 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.
- Aside from explicit loads and stores, the CPU automatically loads instructions from main memory for execution from the location stored in the program counter.
- 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.

Ideally, we want the **programs and data to reside in main memory permanently**. This arrangement usually is not possible on most systems **for two reasons:**

1. Main memory is usually too small to store all needed programs and data permanently.
2. Main memory is volatile—it loses its contents when power is turned off or otherwise lost.

Thus, most computer systems provide **secondary storage a**s an extension of main memory. The main requirement for secondary storage is that it be able to hold large quantities of data permanently.

The most common secondary-storage devices are **hard-disk drives (HDDs) and nonvolatile memory (NVM) devices,** which provide storage for both programs and data. Most programs (system and application) are stored in secondary storage until they are loaded into memory. Many programs then use secondary storage as both the source and the destination of their processing.

In a larger sense, however, the storage structure that we have described —consisting of registers, main memory, and secondary storage—is only one of many possible storage system designs. Other possible components include cache memory, CD-ROM or blu-ray, magnetic tapes, and so on. Those that are slow enough and large enough that they are used only for special purposes — to store backup copies of material stored on other devices, for example— are called **tertiary storage.**

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, size, and volatility.**

The wide variety of storage systems can be organized in a hierarchy (Figure 1.6) according to storage capacity and access time. As a general rule, there is a trade-off between size and speed, with smaller and faster memory closer to the CPU. As shown in the figure, in addition to differing in speed and capacity, the various storage systems are either volatile or nonvolatile.Volatile storage, as mentioned earlier, loses its contents when the power to the device is removed, so data must be written to nonvolatile storage for safekeeping.

The top four levels of memory in the figure are constructed using **semiconductor memory**, which consists of semiconductor-based electronic circuits. NVM devices, at the fourth level, have several variants but in general are faster than hard disks. The most common form of NVM device is flash memory, which is popular in mobile devices such as smartphones and tablets. Increasingly, flash memory is being used for long-term storage on laptops, desktops, and servers as well.

- Volatile storage will be referred to simply as **memory**.
- Nonvolatile storage retains its contents when power is lost. It will be referred to as **NVS**. The vast majority of the time we spend on NVS will be on secondary storage. This type of storage can be classified into two distinct types:
  - **Mechanical**. A few examples of such storage systems are HDDs, optical disks, holographic storage, and magnetic tape.

Electrical. A few examples of such storage systems are flash memory, FRAM, NRAM, and SSD. Electrical storage will be referred to as **NVM**.

Mechanical storage is generally larger and less expensive per byte than electrical storage. Conversely, electrical storage is typically costly, smaller, and faster than mechanical storage.

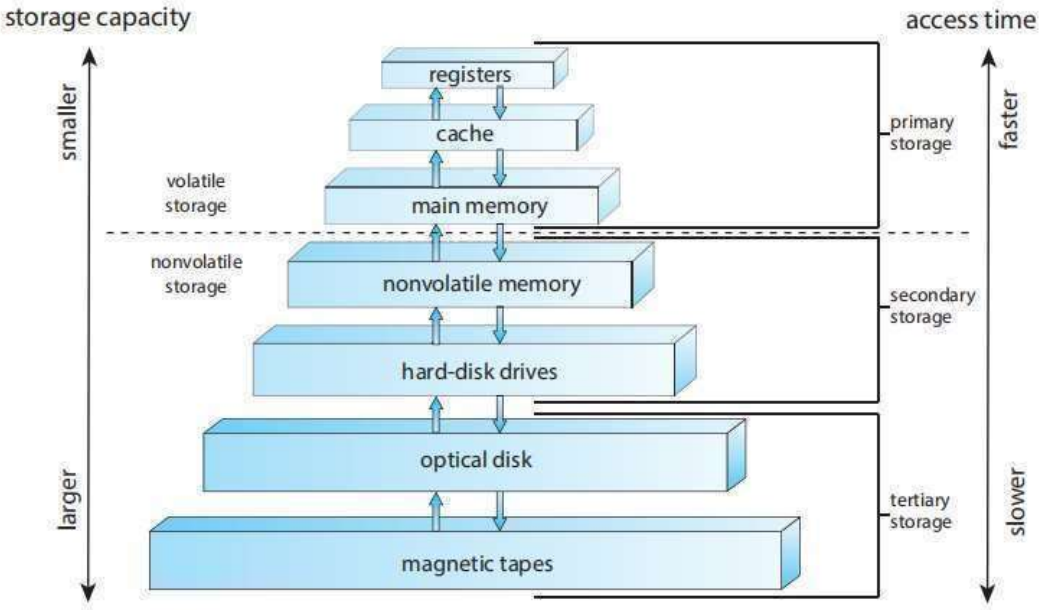


Figure 1.6 Storage-device hierarchy.

Caches can be installed to improve performance where a large disparity in access time or transfer rate exists between two components. aches can be installed to improve performance where a large disparity in access time or transfer rate exists between two components.

I/O Structure

A large portion of operating system code is dedicated to managing I/O, both because of its importance to the reliability and performance of a system and because of the varying nature of the devices.

A general-purpose computer system consists of multiple devices, all of which exchange data via a common bus. The 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 NVS I/O. 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 the device and main 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. In this case, DMA is even more effective. Figure 1.7 shows the interplay of all components of a computer system.

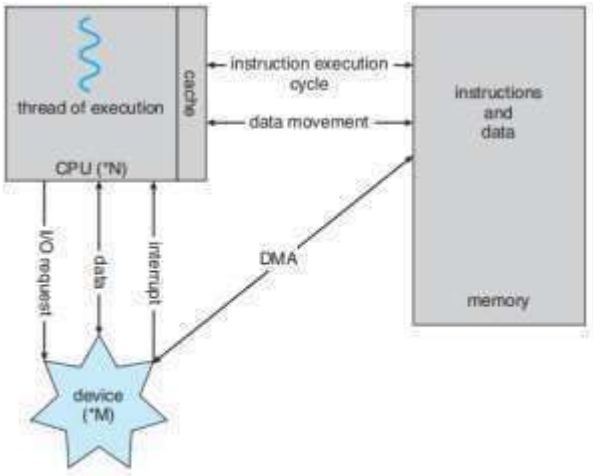


Figure 1.7 How a modern computer system works.

3. Computer-System Architecture

A computer system can be organized in several different ways, which we can categorize roughly according to the number of general-purpose processors used.

1. Single-Processor Systems

- Many years ago, most computer systems used a single processor containing one CPU with a single processing core.

- The **core** is the **component that executes instructions and registers for storing data locally.**

- The one main CPU with its core is capable of executing a general-purpose instruction set, including instructions from processes. These 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. All of these special-purpose processors run a limited instruction set and do not run processes.

- The operating system cannot communicate with these processors; they do their jobs autonomously.

- The use of special-purpose microprocessors is common and does not turn a single-processor system into a multiprocessor.

If there is only one general-purpose CPU with a single processing core, then the system is a single-processor system. Very few contemporary computer systems are single-processor systems.

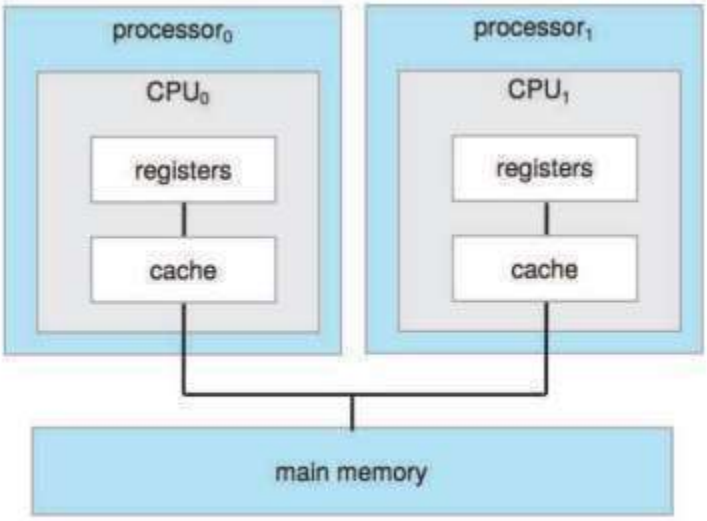


Figure 1.8 Symmetric multiprocessing architecture.

Multiprocessor Systems

- Traditionally, such systems have two (or more) processors, each with a single-core CPU. The processors share the computer bus and sometimes the clock, memory, and peripheral devices. **The primary advantage of multiprocessor systems is increased throughput.**

- That is, by increasing the number of processors, we expect to get more work done in less time. The speed-up ratio with N processors is not N, however; it is less than N.

- When multiple processors cooperate on a task, a certain amount of overhead is incurred in keeping all the parts working correctly. This overhead, plus contention for shared resources, lowers the expected gain from additional processors.

- The most common multiprocessor systems use **symmetric multiprocessing (SMP)**, in which each peer CPU processor performs all tasks, including operating-system functions and user processes.

- Figure 1.8 illustrates a typical SMP architecture with two processors, each with its own CPU. Each CPU processor has its own set of registers, as well as a private—or local— cache.

- However, all processors share physical memory over the system bus.

- The benefit of this model is that many processes can run simultaneously —N processes can run if there are N CPUs—without causing performance to deteriorate significantly. However, since the CPUs are separate, one may be sitting idle while another is overloaded, resulting in inefficiencies.

- These inefficiencies can be avoided if the processors share certain data structures. A multiprocessor system of this form will allow processes and resources—such as memory— to be shared dynamically among the various processors and can lower the workload variance among the processors.

- Figure 1.9 shows a dual-core design with two cores on the same processor chip. In this design, each core has its own register set, as well as its own local cache, often known as a level 1, or L1, cache.

- A level 2 (L2) cache is local to the chip but is shared by the two processing cores. Most architectures adopt this approach, combining local and shared caches, where local, lower-level caches are generally smaller and faster than higher-level shared caches.
- Aside from architectural considerations, such as cache, memory, and bus contention, a multicore processor with N cores appears to the operating system as N standard CPUs.
- Virtually all modern operating systems—including Windows, macOS, and Linux, as well as Android and iOS mobile systems—support multicore SMP systems.
- Adding additional CPUs to a multiprocessor system will increase computng power; however, as suggested earlier, the concept does not scale very well, and once we add too many CPUs, contention for the system bus becomes a bottleneck and performance begins to degrade.
- An alternative approach is instead to provide each CPU (or group of CPUs) with its own local memory that is accessed via a small, fast local bus.
- The CPUs are connected by a shared system interconnect, so that all CPUs share one physical address space. This approach—known as non-uniform memory access, or NUMA—is illustrated in Figure 1.10.
- The advantage is that, when a CPU accesses its local memory, not only is it fast, but there is also no contention over the system interconnect.
- Thus, NUMA systems can scale more effectively as more processors are added. A potential drawback with a NUMA system is increased latency when a CPU must access remote memory across the system interconnect, creating a possible performance penalty.
- Because NUMA systems can scale to accommodate a large number of processors, they are becoming increasingly popular on servers as well as high-performance computing systems.

Reference Video	
	Multiprocessing OS https://youtu.be/IZfWjg3U3mA

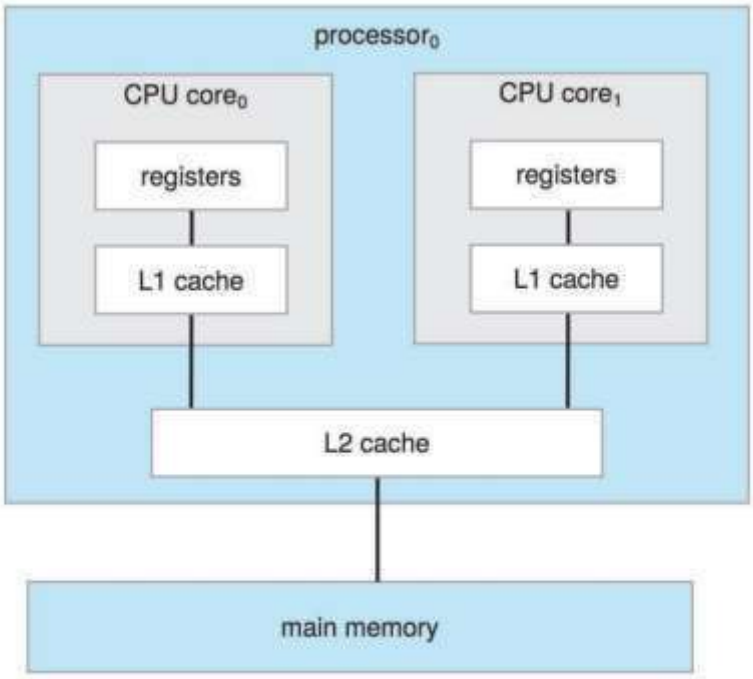


Figure 1.9 A dual-core design with two cores on the same chip.

- **Blade servers** are systems in which multiple processor boards, I/O boards, and networking boards are placed in the **same chassis**.

- The difference between these and traditional multiprocessor systems is that each bladeprocessor board boots independently and runs its own operating system. Some blade-server boards are multiprocessor as well, which blurs the lines between types of computers. In essence, these servers consist of multiple independent multiprocessor systems.

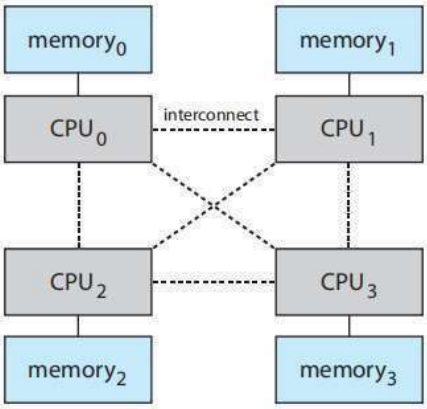


Figure 1.10 NUMA multiprocessing architecture.

Clustered Systems

A **clustered system** is a type of multiprocessor system, which gathers together multiple CPUs. Clustered systems differ from the multiprocessor systems in that they are composed of ***two or more individual systems***—or ***nodes—joined together***; each node is typically a multicore system. Such systems are considered **loosely coupled**.

Clustered computers share storage and are closely linked via a local-area network LAN or a faster interconnect, such as InfiniBand. Clustering is usually used to provide **high-availability service**— that is, service that will continue even if one or more systems in the cluster fail.

A layer of cluster software runs on the cluster nodes. Each node can monitor one or more of the others (over the network). 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. High availability provides increased reliability, which 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.

Clustering can be structured asymmetrically or symmetrically. In **asymmetric clustering,** one machine is in hot-standby mode while the other is running the applications. The hot-standby host machine does nothing but monitor the active server. If that server fails, the hot-standby host becomes the active server.

In **symmetric clustering**, two or more hosts are running applications and are monitoring each other. This structure is obviously more efficient, as it uses all of the available hardware. However, it does require that more than one application be available to run.n parallel on individual cores in a computer or computers in a cluster.

Each machine has full access to all data in the database. To provide this shared access, the system must also supply access control and locking to ensure that no conflicting operations occur. This function, commonly known as a **distributed lock manager (DLM)**, is included in some cluster technology.

Cluster technology is changing rapidly. Some cluster products support thousands of systems in a cluster, as well as clustered nodes that are separated by miles. Many of these improvements are made possible by **storage-area networks (SANs),** which allow many systems to attach to a pool of storage. If the applications and their data are stored on the SAN, then the cluster software can assign the application to run on any host that is attached to the SAN. If the host fails, then any other host can take over. In a database cluster, dozens of hosts can share the same database, greatly increasing performance and reliability. Figure 1.11 depicts the **general structure of a clustered system.**

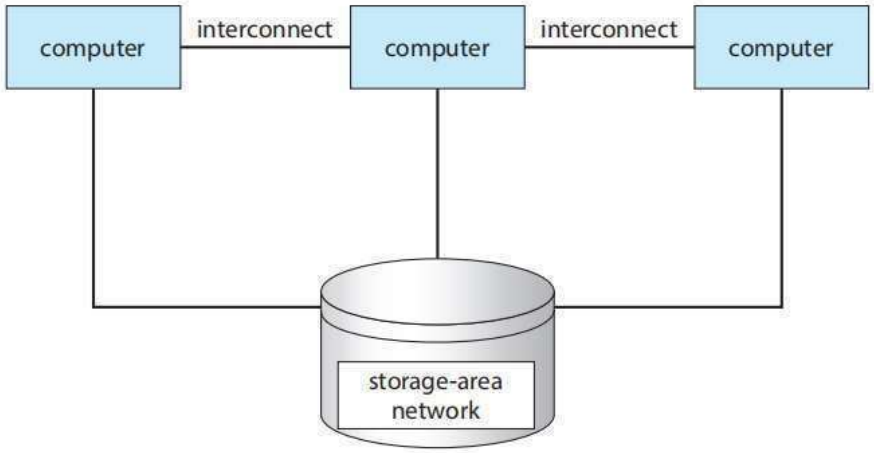


Figure 1.11 General structure of a clustered system.

Reference Video	
	Stoarage Area Networks (SAN) https://youtu.be/Pu4b8K0BQ9Y

In a multiprogrammed system, a **program in execution is termed a process.** The operating system keeps several processes in memory simultaneously (Figure 1.12). **The operating system picks and begins to execute one of these processes.** Eventually, the process 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 o**perating system simply switches to, and executes, another process.** When that process needs to wait, the CPU switches to another process, and so on. Eventually, the first process finishes waiting and gets the CPU back. As long as at least one process needs to execute, the CPU is never idle.

Multitasking is a logical extension of multiprogramming. In multitasking systems, the CPU executes multiple processes by switching among them, but the switches occur frequently, providing the user with a fast response time.

Dual-Mode and Multimode Operation

In order to ensure the proper execution of the operating system, there must be able to **distinguish between the execution of operating-system code and user defined code.**

There are 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 fulfill the request. This is shown in Figure 1.13.

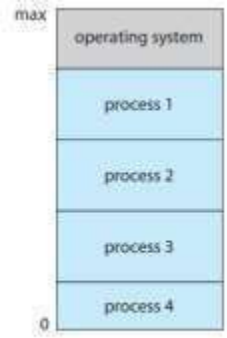


Figure 1.12 Memory layout for a multiprogramming system.

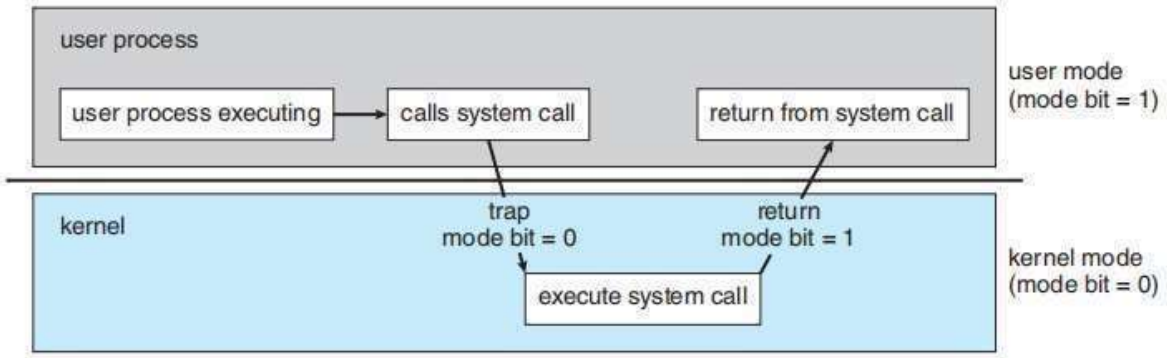


Figure 1.13 Transition from user to kernel mode.

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.** 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.

The instruction to switch to kernel mode is an example of a privileged instruction. Some other examples include I/O control, timer management, and interrupt

CPUs that support virtualization frequently have a separate mode to indicate when the virtual machine manager (VMM)—and the virtualization management software—is in control of the system. In this mode, the VMM has more privileges than user processes but fewer than the kernel

The lack of a hardware-supported dual mode can cause serious shortcomings in an operating system. Once hardware protection is in place, it detects errors that violate modes. These errors are normally handled by the operating system.

- **Timer:**

We must ensure that 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 i**s 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 occur.

Reference Video
User Mode and Kernel Mode in Windows https://youtu.be/RK8mRIf5bMg

Resource Management

An operating system is a resource manager. The system's CPU, memory space, file-storage space, and I/O devices are among the resources that the operating system must manage.

Process Management

A program can do nothing unless its instructions are executed by a CPU. **A program in execution is a process.** A program such as a compiler is a **process**, and a word-processing program being run by an individual user on a PC is a process. Similarly, a social media app on a mobile device is a process. It is possible to provide system calls that allow processes to create subprocesses to execute concurrently.

A process needs certain **resources—including CPU time, memory, files, and I/O devices—** to accomplish its task. These resources are typically allocated to the process while **it is running**. In addition to the various physical and logical resources that a process obtains when it is created, various initialization data (input) may be passed along.

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 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 core—or in parallel across multiple CPU cores.

The operating system is responsible for the following activities in connection with process management:

- Creating and deleting both user and system processes
- Scheduling processes and threads on the CPUs
- Suspending and resuming processes
- Providing mechanisms for process synchronization
- Providing mechanisms for process communication

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 CPU reads instructions from main memory during the instruction-fetch cycle and both reads and writes data from main memory during the data-fetch cycle (on a von Neumann architecture). The main memory is generally the only large storage device that the CPU is able to address and access directly. Instructions must be in memory for the CPU to execute them. The operating system is responsible for the following activities in connection with memory management:

- Keeping track of which parts of memory are currently being used and which process is using them
- Allocating and deallocating memory space as needed
- Deciding which processes (or parts of processes) and data to move into and out of memory

File-System Management

The operating system abstracts from the physical properties of its storage devices to define a logical storage unit, the **file**. The operating system maps files onto physical media and accesses these files via the storage devices.

Each of these media has its own characteristics and physical organization. Most are controlled by a device, such as a disk drive, that also has its own unique characteristics. These properties include access speed, capacity, data-transfer rate, and access method (sequential or random).

A **file** is a ***collection of related information defined by its creator.*** Commonly, files **represent programs** (both source and object forms) **and data**. Data files may be numeric, alphabetic, alphanumeric, or binary. Files may be free form (for example, text files), or they may be formatted rigidly (for example, fixed fields such as an mp3 music file). Clearly, the concept of a file is an extremely general one.

The operating system implements the abstract concept of a file by managing mass storage media and the devices that control them. In addition, files are normally **organized into directorie**s to make them easier to use. Finally, when multiple users have access to files, it may be desirable to control which user may access a file and how that user may access it (for example, read, write, append).

The operating system is responsible for the following activities in connection with file management:

- Creating and deleting files
- Creating and deleting directories to organize files
- Supporting primitives for manipulating files and directories
- Mapping files onto mass storage
- Backing up files on stable (nonvolatile) storage media

Mass-Storage Management

The computer system **must provide secondary storage to back up main memory.** Most modern computer systems use HDDs and NVM devices as the principal on-line storage media for both programs and data. Most programs—including compilers, web browsers, word processors, and games—are stored on these devices until loaded into memory. The programs then use the devices as both the source and the destination of their processing.

Hence, the proper management of secondary storage is of central importance to a computer system. **The operating system is responsible for the following activities in connection with secondary storage management:**

- Mounting and unmounting
- Free-space management
- Storage allocation
- Disk scheduling

- Partitioning
- Protection

Cache Management

Caching is an important principle of computer systems. Information is normally kept in some storage system (such as main memory). As it is used, it is **copied into a faster storage system— the cache—on a temporary basis.** When we need a particular piece of information, we first check whether it is in the cache. If it is, we use the information directly from the cache.

Internal programmable registers provide a **high-speed cach**e for main memory. The programmer (or compiler) implements the register allocation and register-replacement algorithms to decide which information to keep in registers and which to keep in main memory. **Other caches are implemented totally in hardware.**

For instance, most systems have an instruction cache to hold the instructions expected to be executed next. Without this cache, the CPU would have to wait several cycles while an instruction was fetched from main memory. Careful selection of the cache size and of a replacement policy can result in greatly increased performance, as you can see by examining Figure 1.14.

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

Figure 1.14 Characteristics of various types of storage.

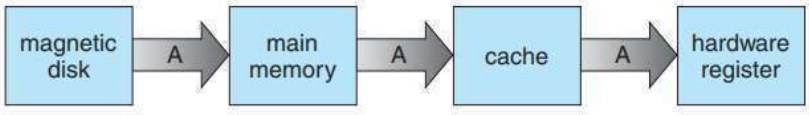


Figure 1.15 Migration of integer A from disk to register.

The movement of information between levels of a storage hierarchy may be either **explicit** or **implicit**, depending on the hardware design and the controlling operating-system software. For instance, data transfer from cache to CPU and registers is usually a hardware function, with no operating-system intervention.

The copy of A appears in several places: on the hard disk, in main memory, in the cache, and in an internal register (see Figure 1.15). 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 hard disk.

Since the various CPUs can all execute in parallel, one must make sure that an update to the value of variable in one cache is immediately reflected in all other caches where the variable resides. This situation is called **cache coherency,** and it is usually a hardware issue (handled below the operating-system level).

I/O System Management

One of the purposes of an operating system is to hide the peculiarities of specific hardware devices from the user. **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
- Only the device driver knows the peculiarities of the specific device to which it is assigned.

Security and Protection

Protection is any **mechanism for controlling the access of processes or users to the resources defined by a computer system**. This mechanism must provide means to specify the controls to be imposed and to enforce the controls. Protection can improve reliability by detecting latent errors at the interfaces between component subsystems. Early detection of interface errors can often prevent contamination of a healthy subsystem by another subsystem that is malfunctioning. Furthermore, an unprotected resource cannot defend against use (or misuse) by an unauthorized or incompetent user.

A **protection-oriented system** provides a means to distinguish between authorized and unauthorized usage. A system can have adequate protection but still be prone to failure and allow inappropriate access.

T**he job of security to defend a system from external and internal attacks.** Such attacks spread across a huge range and include viruses and worms, denial-of service attacks (which use all of a system's resources and so keep legitimate users out of the system), identity theft, and theft of service (unauthorized use of a system).

Prevention of some of these attacks is considered an operating system function on some systems, while other systems leave it to policy or additional software. Due to the alarming rise in security incidents, operating system security features are a fast-growing area of research and implementation.

Protection and security require the system to be able to distinguish among all its users. Most operating systems maintain a list of user names and associated **user identifier (user IDs)**. In Windows parlance, this is a **security ID (SID).** These numerical IDs are unique, one per user.

Security and Protection

When a user logs in to the system, the authentication stage determines the appropriate user ID for the user. That user ID is associated with all of the user's processes and threads. When an ID needs to be readable by a user, it is translated back to the user name via the user name list. In UNIX systems, Group functionality can be implemented as a system-wide list of group names and **group identifier** .

A user can be in one or more groups, depending on operating-system design decisions. The user's group IDs are also included in every associated process and thread. In the course of normal system use, the user ID and group ID for a user are sufficient. However, a user sometimes needs to **escalate privileges to gain extra permissions for an activity**. The user may need access to a device that is restricted, for example. Operating systems provide various methods to allow privilege escalation. On UNIX, for instance, the setuid attribute on a program causes that program to run with the user ID of the owner of the file, rather than the current user's ID. The process runs with this **effective UID** until it turns off the extra privileges or terminates.

Reference Video	
	**Operating System Security** https://youtu.be/qb_rBR8DkPE

Virtualization

Virtualization is a technology that allows us to **abstract the hardware of a single computer** (the CPU, memory, disk drives, network interface cards, and so forth) into **several different execution environments, thereby creating the illusion that each separate environment is running on its own private computer.**

These environments can be **viewed as different individual operating systems (for example, Windows and UNIX) that may be running at the same time and may interact with each other.** A user of a virtual machine can switch among the various operating systems in the same way a user can switch among the various processes running concurrently in a single operating system.

Virtualization allows operating systems to run as applications within other operating systems. At first blush, there seems to be little reason for such functionality. But the virtualization industry is vast and growing, which is a testament to its utility and importance. Broadly speaking, virtualization software is one member of a class that also includes emulation.

Emulation, which involves simulating computer hardware in software, is typically used when the source CPU type is different from the target CPU type. For example, when Apple switched from the IBM Power CPU to the Intel x86 CPU for its desktop and laptop computers, it included an emulation facility called "Rosetta," which allowed applications compiled for the IBM CPU to run on the Intel CPU. That same concept can be extended to allow an entire operating system written for one platform to run on another. Emulation comes at a heavy price, however.

Every machine-level instruction that runs natively on the source system must be translated to the equivalent function on the target system, frequently resulting in several target instructions. If the source and target CPUs have similar performance levels, the emulated code may run much more slowly than the native code.

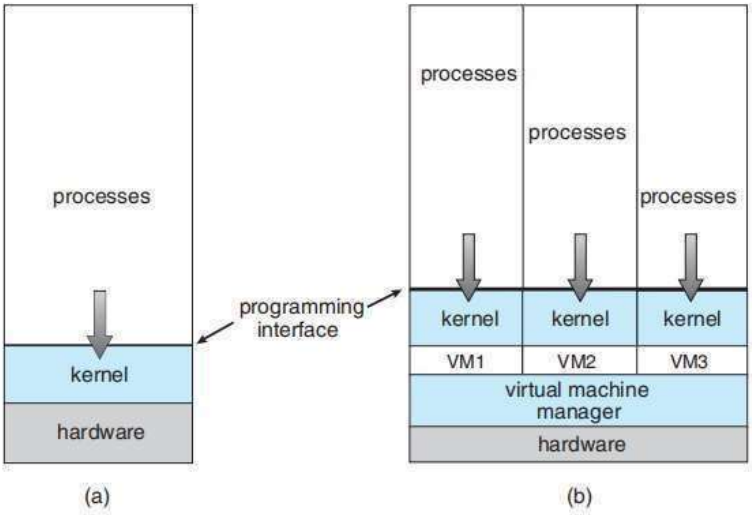


Figure 1.16 A computer running (a) a single operating system and (b) three virtua machines.

With virtualization, in contrast, an operating system that is natively compiled for a particular CPU architecture runs within another operating system also native to that CPU. Virtualization first came about on IBM mainframes as a method for multiple users to run tasks concurrently.

Running multiple virtual machines allowed (and still allows) many users to run tasks on a system designed for a singe user. Later, in response to problems with running multiple Microsoft Windows applications on the Intel x86 CPU, VMware created a new virtualization technology in the form of an application that ran on Windows. That application ran one or more **guest** copies of Windows or other native x86 operating systems, each running its own applications. (See Figure 1.16.).

Windows was the host operating system, and the VMware application was the virtual machine manager (VMM). The VMM runs the guest operating systems, manages their resource use, and protects each guest from the others. Even though modern operating systems are fully capable of running multiple applications reliably, the use of virtualization continues to grow. On laptops and desktops, a VMM allows the user to install multiple operating systems for exploration or to run applications written for operating systems other than the native host.

Companies writing software for **multiple operating systems can use virtualization to run all of those operating systems on a single physical server for development, testing, and debugging**. Within data centers, virtualization has become a common method of executing and managing computing environments. VMMs like VMware ESX and Citrix XenServer no longer run on host operating systems but rather are the host operating systems, providing services and resource management to virtual machine processes.

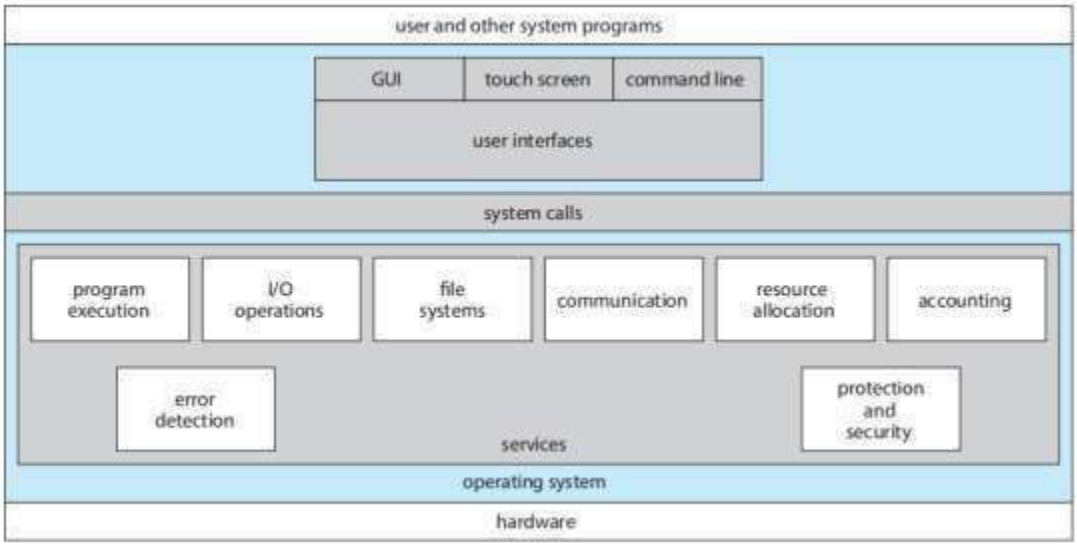


Figure 1.17 - A view of Operating System Services

Reference Video	
	Virtualization https://youtu.be/iBI31dmqSX0

Operating-System Services

An operating system provides an environment for the execution of programs. It makes certain services available 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.

Figure 2.1 shows **one view of the various operating-system services and how they interrelate**. Note that these services also make the programming task easier for the programmer. One set of operating system services provides functions that are helpful to the user.

- **User interface.**

- Almost all operating systems have a user interface (UI). This interface can take several forms.
    - Most commonly, a **graphical user interface (GUI)** is used. Here, the interface is a window system with a mouse that serves as a pointing device to direct I/O, choose from menus, and make selections and a keyboard to enter text.
    - Mobile systems such as phones and tablets provide a **touch-screen interface**, enabling users to slide their fingers across the screen or press buttons on the screen to select choices.
    - Another option 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). 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 reading from a network interface or writing to a file system).
  - ➢ 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**
  - ➢ **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.**
  - ➢ There are many circumstances in which **one process needs to exchange information with another process**.
  - ➢ Such 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 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 or an attempt to access an illegal memory location).

➢ For each type of error, the operating system should take the appropriate action to ensure correct and consistent computing.

- **Resource allocation.**
  - ➢ When there are multiple processes 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.
  - ➢ There may also be routines to allocate printers, USB storage drives, and other peripheral devices.

- **Logging.**
  - ➢ We want to keep track of which programs 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 system administrators 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 himself or herself to the system, usually by means of a password, to gain access to system resources.

➢ It extends to defending external I/O devices, including network adapters, from invalid access attempts and recording all such connections for detection of break-ins. 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.

Reference Video	
	**Structure of Operating System** https://youtu.be/XXPBl20J22w

User and Operating-System Interface

- The approaches for users to interface with the operating system are - **command-line interface**, or **command interpreter**, that allows users to directly enter commands to be performed by the operating system. The other two allow users to interface with the operating system via a **graphical user interface**, or GUI.

Command Interpreters

- Most operating systems, including Linux, UNIX, and Windows, **treat the command interpreter as a special program that is running when a process is initiated or when a user first logs on** (on interactive systems). On systems with multiple command interpreters to choose from, the interpreters are known as shells.
- For example, on UNIX and Linux systems, **a user may choose among several different shells, including the C shell, Bourne-Again shell, Korn shell, and others.** Third-party shells and free user-written shells are also available.
- Most shells provide similar functionality, and a user's choice of which shell to use is generally based on personal preference. Figure 1.18 shows the Bourne-Again (or bash) shell command interpreter being used on macOS.
- The **main function of the command interpreter is to get and execute the next user-specified command.** Many of the commands given at this level manipulate files: create, delete, list, print, copy, execute, and so on. The various shells available on UNIX systems operate in this way.

<u>These commands can be implemented in two general ways.</u>

- ➢ The command interpreter itself contains the code to execute the command.
- ➢ Operating system implements most commands through system programs.

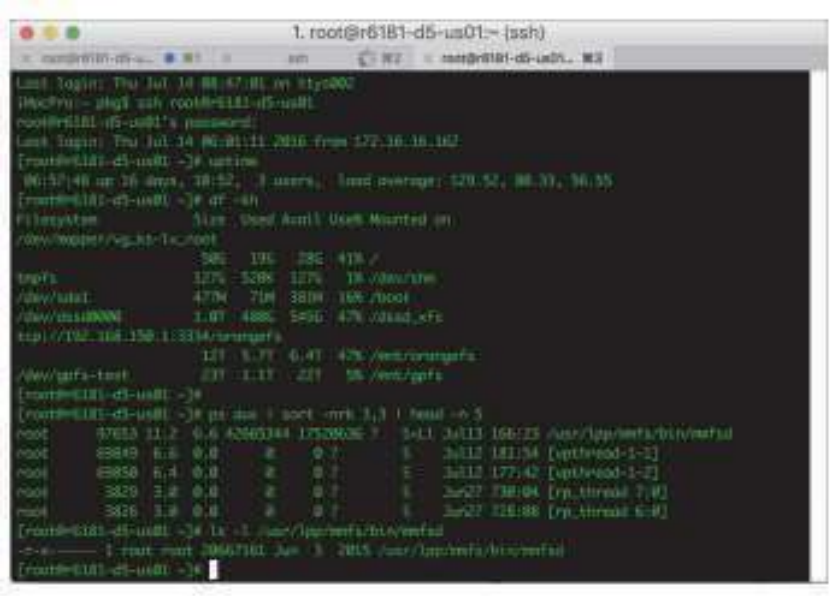


Figure 1.18 - The bash shell command interpreter in macOS

Graphical User Interface

A 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.

The **first GUI appeared** on the Xerox Alto computer in 1973. However, graphical interfaces became more widespread with the advent of Apple Macintosh computers in the 1980s. The user interface for the Macintosh operating system has undergone various changes over the years, the most significant being the adoption of the **Aqua** interface that appeared with macOS.

Microsoft's first version of **Windows— Version 1.0—**was based on the addition of a GUI interface to the MS-DOS operating system.

Traditionally, UNIX systems have been dominated by command-line interfaces. Various GUI interfaces are available with significant development in GUI designs from various open-source projects, such as **K Desktop Environment (or KDE) and the GNOME** desktop by the GNU project. Both the KDE and GNOME desktops run on Linux and various UNIX systems and are available under open-source licenses, which means their source code is readily available for reading and for modification under specific license terms.

Touch-Screen Interface

Because a either a command-line interface or a mouse-and-keyboard system is impractical for most mobile systems, smartphones and handheld tablet computers typically use a touch-screen interface. Here, users interact by **making gestures on the touch screen— for example, pressing and swiping fingers across the screen**. Although earlier smartphones included a physical keyboard, most smartphones and tablets now simulate a keyboard on the touch screen.

Figure 1.19 illustrates the touch screen of the Apple iPhone. Both the iPad and the iPhone use the Springboard touch-screen interface.

Figure 1.19 - The iPhone 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. For them, it is more efficient, giving them faster access to the activities they need to perform.

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. 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.

In contrast, most Windows users are happy to use the Windows GUI environment and almost never use the shell interface. Recent versions of the Windows operating system provide both a standard GUI for desktop and traditional laptops and a touch screen for tablets. The various changes undergone by the Macintosh operating systems also provide a nice study in contrast. Historically, Mac OS has not provided a command-line interface, always requiring its users to interface with the operating system using its GUI. However, with the release of macOS (which is in part implemented using a UNIX kernel), the operating system now provides both an Aqua GUI and a command-line interface.

Almost all users of mobile systems interact with their devices using the touch-screen interface. The user interface can vary from system to system and even from user to user within a system; however, it typically is substantially removed from the actual system structure. The design of a useful and intuitive user interface is therefore not a direct function of the operating system.

System calls

System calls **provide an interface to the services made available by an operating system**. These calls are generally available as functions written in C and C++, although certain low-level tasks may have to be written using assembly-language instructions.

Example

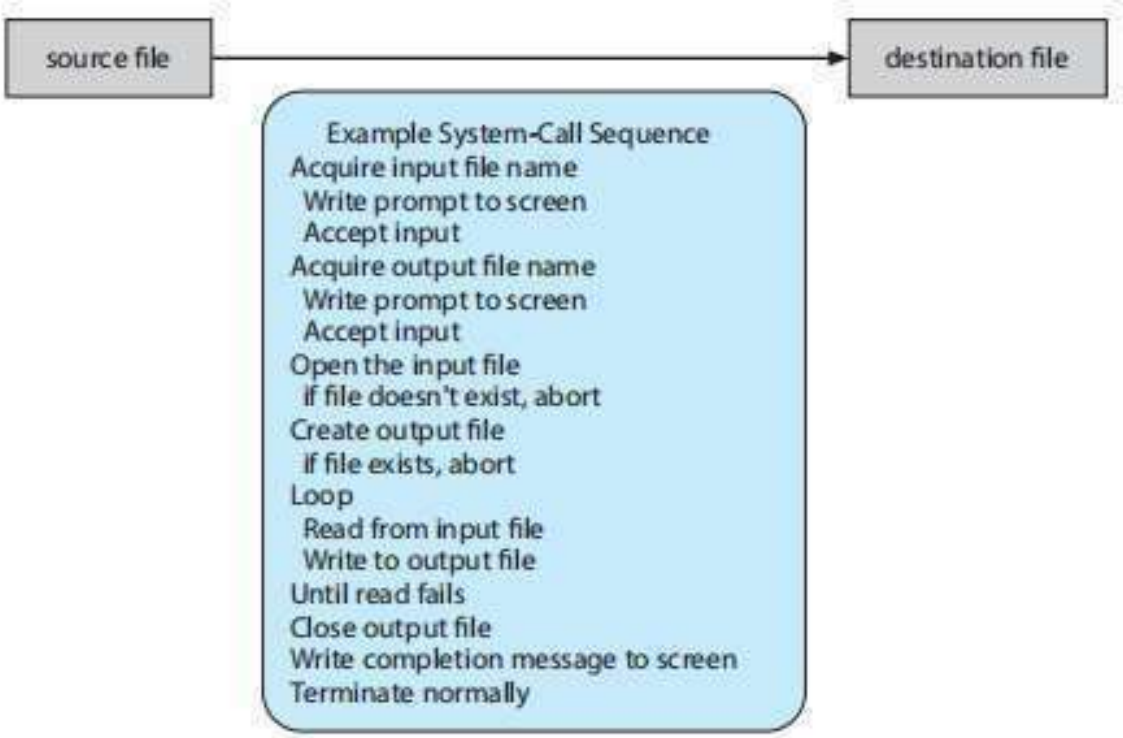
Let's use an example to learn how system calls are used: writing a simple program to read data from one file and copy them to another file. The first input that the program will need is the names of the two files: the input file and the output file.

These names can be specified in many ways, depending on the operating-system design. One approach is to pass the names of the two

files as part of the command— for example, the UNIX cp command:

cp in.txt out.txt

This command copies the input file in.txt to the output file out.txt. A second approach is for the program to ask the user for the names.



Application Programming Interface

Even simple programs may make heavy use of the operating system. Frequently, systems execute thousands of system calls per second. Most programmers never see this level of detail, however. Typically, application developers design programs according to an **application programming interface (API).** The API s**pecifies a set of functions that are available to an application programmer, including the parameters that are passed to each function and the return values the programmer can expect.**

Three of the most common APIs available to application programmers are the Windows API for Windows systems, the POSIX API for POSIX-based systems (which include virtually all versions of UNIX, Linux, and macOS), and the Java API for programs that run on the Java virtual machine.

A programmer accesses an API via a library of code provided by the operating system. In the case of UNIX and Linux for programs written in the C language, the library is called **libc**. Note that—unless specified — the system-call names used throughout this text are generic examples. Each operating system has its own name for each system call.

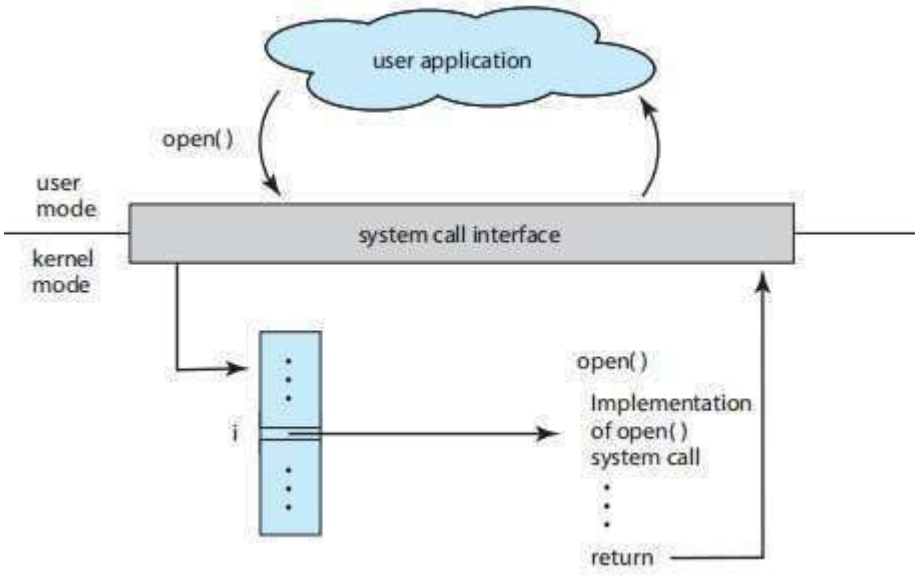


Figure 1.20 - The handling of a user application onvoking the open() system call

Another important factor in handling system calls is the **run-time environment (RTE)**— the full suite of software needed to execute applications written in a given programming language, including its compilers or interpreters as well as other software, such as libraries and loaders. The RTE provides a **system-call interface** that serves as the link to system calls made available by the operating system. The system-call interface intercepts function calls in the API and invokes the necessary system calls within the operating system.

Reference Video	
	**System Calls** https://youtu.be/lhToWeuWWfw

Types of System Calls

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

- Process control
  - create process, terminate process
  - load, execute
  - get process attributes, set process attributes
  - 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
- Protection
  - get file permissions
  - set file permissions

EXAMPLES OF WINDOWS AND UNIX SYSTEM CALLS

The following illustrates various equivalent system calls for Windows and UNIX operating systems.

	Windows	Unix

Process control	`CreateProcess()`
	`ExitProcess()`
	`WaitForSingleObject()`
		`fork()`
		`exit()`
		`wait()`

File management	`CreateFile()`
	`ReadFile()`
	`WriteFile()`
	`CloseHandle()`
		`open()`
		`read()`
		`write()`
		`close()`

Device management	`SetConsoleMode()`
	`ReadConsole()`
	`WriteConsole()`
		`ioctl()`
		`read()`
		`write()`

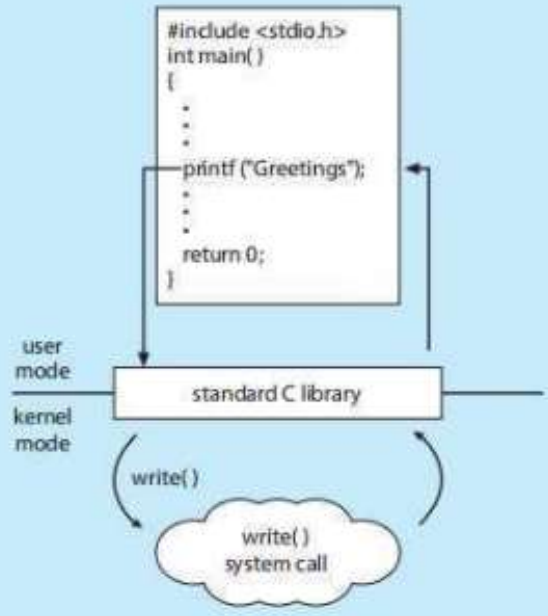
Information maintenance	`GetCurrentProcessID()`
	`SetTimer()`
	`Sleep()`
		`getpid()`
		`alarm()`
		`sleep()`

Communications	`CreatePipe()`
	`CreateFileMapping()`
	`MapViewOfFile()`
		`pipe()`
		`shm_open()`
		`mmap()`

Protection	`SetFileSecurity()`
	`InitlializeSecurityDescriptor()`
	`SetSecurityDescriptorGroup()`
		`chmod()`
		`umask()`
		`chown()`

THE STANDARD C LIBRARY

The standard C library provides a portion of the system-call interface for many versions of UNIX and Linux. As an example, let's assume a C program invokes the `printf()` statement. The C library intercepts this call and invokes the necessary system call (or calls) in the operating system—in this instance, the `write()` system call. The C library takes the value returned by `write()` and passes it back to the user program:



System Services

Another aspect of a modern system is its collection of system services. At the lowest level is **hardware**. Next is the operating system, then the system services, and finally the application programs. System services, also known as system utilities, provide a convenient environment for program development and execution. Some of them are simply user interfaces to system calls. Others are considerably more complex. They can be divided into these categories:

File management. These programs create, delete, copy, rename, print, list, and generally access and manipulate files and directories.

Status information. Some programs simply ask the system for the date, time, amount of available memory or disk space, number of users, or similar status information. Others are more complex, providing detailed performance, logging, and debugging information. Typically, these programs format and print the output to the terminal or other output devices or files or display it in a window of the GUI. Some systems also support a registry, which is used to store and retrieve configuration information.

File modification Several text editors may be available to create and modify the content of files stored on disk or other storage devices. There may also be special commands to search contents of files or perform transformations of the text.

Programming-language support. Compilers, assemblers, debuggers, and interpreters for common programming languages (such as C, C++, Java, and Python) are often provided with the operating system or available as a separate download.

Program loading and execution. Once a program is assembled or compiled, it must be loaded into memory to be executed. The system may provide absolute loaders, relocatable loaders, linkage editors, and overlay loaders. Debugging systems for either higher-level languages or machine language are needed as well.

Communications. These programs provide the mechanism for creating virtual connections among processes, users, and computer systems. They allow users to send messages to one another's screens, to browse web pages, to send e-mail messages, to log in remotely, or to transfer files from one machine to another.

Background services. All general-purpose systems have methods for launching certain system-program processes at boot time. Some of these processes terminate after completing their tasks, while others continue to run until the system is halted. Constantly running system-program processes are known as services, subsystems, or daemons.

Typical systems have dozens of daemons. In addition, operating systems that run important activities in user context rather than in kernel context may use daemons to run these activities. Along with system programs, most operating systems are supplied with programs that are useful in solving common problems or performing common operations. Such application programs include web browsers, word processors and text formatters, spreadsheets, database systems, compilers, plotting and statistical-analysis packages, and games.

12. Operating-System Design and Implementation

Although there are some problems in designing and implementing an operating system, There are **no complete solutions to such problems, but there are approaches that have proved successful**.

1. Design Goals

- The first problem in designing a system is to define goals and specifications. At the highest level, the design of the system will be affected by the choice of hardware and the type of system: traditional desktop/laptop, mobile, distributed, or real time.

- Beyond this highest design level, the requirements may be much harder to specify.

- The requirements can be divided into two basic groups: **user goals** and **system goals.**

- **Users want certain obvious properties in a system. The system should be convenient to use, easy to learn and to use, reliable, safe, and fast.**

- These specifications are not particularly useful in the system design, since there is no general agreement on how to achieve them.

- **A similar set of requirements can be defined by the developers who must design, create, maintain, and operate the system.** The system should be easy to design, implement, and maintain; and it should be flexible, reliable, error free, and efficient. Again, these requirements are vague and may be interpreted in various ways.

- There is, in short, no unique solution to the problem of defining the requirements for an operating system. The wide range of systems in existence shows that different requirements can result in a large variety of solutions for different environments.

Mechanisms and Policies

- One important principle is the separation of policy from mechanism.
- **Mechanisms determine how to do something; policies determine what will be done.**
- The separation of policy and mechanism is important for flexibility.
- Policies are likely to change across places or over time. In the worst case, each change
- in policy would require a change in the underlying mechanism.
- A general mechanism flexible enough to work across a range of policies is preferable.
- A change in policy would then require redefinition of only certain parameters of the system.
- If the mechanism is properly separated from policy, it can be used either to support a policy decision that I/O-intensive programs should have priority over CPU-intensive ones or to support the opposite policy.
- Microkernel-based operating systems take the separation of mechanism and policy to one extreme by implementing a basic set of primitive building blocks.
- Whenever the question is how rather than what, it is a mechanism that must be determined.

Implementation

- **Once an operating system is designed, it must be implemented.** Because operating systems are collections of many programs, written by many people over a long period of time, it is difficult to make general statements about how they are implemented.
- **Early operating systems were written in assembly language. Now, most are written in higher-level languages such as C or C++, with small amounts of the system written in assembly language.**

- In fact, more than one higher level language is often used. The lowest levels of the kernel might be written in assembly language and C. Higher-level routines might be written in C and C++, and system libraries might be written in C++ or even higher-level languages.

- Android provides a nice example: **its kernel is written mostly in C with some assembly language**. Most Android system libraries are written in C or C++, and its application frameworks—which provide the developer interface to the system—are written mostly in Java.

- The advantages of using a higher-level language, or at least a systems implementation language, for implementing operating systems are the same as those gained when the language is used for application programs: the code can be written faster, is more compact, and is easierto understand and debug.

- In addition, improvements in compiler technology will improve the generated code for the entire operating system by simple recompilation.

- Finally, an operating system is far easier to port to other hardware if it is written in a higher-level language.

The only possible disadvantages of implementing an operating system in a higher-level language are reduced speed and increased storage requirements.

Reference Video	
	**OS Design and Implementation** https://youtu.be/acPvdGOYK4I

13. Building and Booting an Operating System

It is possible to design, code, and implement an operating system specifically for one specific machine configuration. More commonly, however, operating systems are designed to run on any of a class of machines with a variety of peripheral configurations.

1. Operating-System Generation

- Most commonly, a computer system, when purchased, has an operating system already installed. For example, you may purchase a new laptop with Windows or macOS preinstalled.

- But suppose you wish to replace the preinstalled operating system or add additional operating systems. Or suppose when one purchase a computer without an operating system. Then there are few options for placing the appropriate operating system on the computer and configuring it for use.

While generating (or building) an operating system from scratch, these steps need to follow:

1. Write the operating system source code (or obtain previously written source code).
2. Configure the operating system for the system on which it will run.
3. Compile the operating system.
4. Install the operating system.
5. Boot the computer and its new operating system.

Configuring the system involves specifying which features will be included, and this varies by operating system. Typically, parameters describing how the system is configured is stored in a configuration file of some type, and once this file is created, it can be used in several ways.

To build a Linux system from scratch, it is typically necessary to perform the following steps:

1. Download the Linux source code from http://www.kernel.org.
2. Configure the kernel using the "make menuconfig" command. This step generates the .config configuration file.
3. Compile the main kernel using the "make" command. The make command compiles the kernel based on the configuration parameters identified in the .config file, producing the file vmlinuz, which is the kernel image.
4. Compile the kernel modules using the "make modules" command. Just as with compiling the kernel, module compilation depends on the configuration parameters specified in the .config file.
5. Use the command "make modules install" to install the kernel modules into vmlinuz.
6. Install the new kernel on the system by entering the "make install" command.

When the system reboots, it will begin running this new operating system.

Alternatively, it is possible to modify an existing system by installing a

Linux virtual machine. This will allow the host operating system (such as

Windows or macOS) to run Linux.

To build the operating system used in the virtual machine :

1. Downloaded the Ubuntu ISO image from https://www.ubuntu.com/
2. Instructed the virtual machine software VirtualBox to use the ISO as the bootable medium and booted the virtual machine
3. Answered the installation questions and then installed and booted the operating system as a virtual machine

1.13.2 System Boot

After an operating system is generated, it must be made available for use by the hardware. But how does the hardware know where the kernel is or how to load that kernel? **The process of starting a computer by loading the kernel is known as booting the system**. On most systems, the **boot process** proceeds as follows:

1. A small piece of code known as the bootstrap program or boot loader

locates the kernel.

2. The kernel is loaded into memory and started.
3. The kernel initializes hardware.
4. The root file system is mounted.

Some computer systems use a **multistage boot process**: When the computer is first powered on, a small boot loader located in nonvolatile firmware known as BIOS is run. This initial boot loader usually does nothing more than load a second boot loader, which is located at a fixed disk location called the boot block.

The program stored in the boot block may be sophisticated enough to load the entire operating system into memory and begin its execution. More typically, it is simple code (as it must fit in a single disk block) and knows only the address on disk and the length of the remainder of the bootstrap program.

Many recent computer systems have replaced the **BIOS-based boot process with UEFI (Unified Extensible Firmware Interface)**. UEFI has several advantages over BIOS, including better support for 64-bit systems and larger disks. Perhaps the greatest advantage is that UEFI is a single, complete boot manager and therefore is faster than the multistage BIOS boot process.

Whether booting from BIOS or UEFI, the bootstrap program can perform a variety of tasks. In addition to loading the file containing the kernel program into memory, it also runs diagnostics to determine the state of the machine — for example, inspecting memory and the CPU and discovering devices.

If the diagnostics pass, the program can continue with the booting steps. The bootstrap can also initialize all aspects of the system, from CPU registers to device controllers and the contents of main memory. Sooner or later, it starts the operating system and mounts the root file system. It is only at this point is the system said to be running.

GRUB is an o**pen-source bootstrap program for Linux and UNIX systems.** Boot parameters for the system are set in a GRUB configuration file, which is loaded at startup. GRUB is flexible and allows changes to be made at boot time, including modifying kernel parameters and even selecting among different kernels that can be booted.

As an example, the following are kernel parameters from the special Linux file /proc/cmdline, which is used at boot time:

BOOT IMAGE=/boot/vmlinuz-4.4.0-59-generic root=UUID=5f2e2232-4e47-4fe8-ae94-45ea749a5c92

BOOT IMAGE is the name of the kernel image to be loaded into memory, and root specifies a unique identifier of the root file system.

To save space as well as decrease boot time, the Linux kernel image is a compressed file that is extracted after it is loaded into memory. During the boot process, the boot loader typically creates a temporary RAM file system, known as initramfs.

Finally, boot loaders for most operating systems—including Windows, Linux, and macOS, as well as both iOS and Android—**provide booting into recovery mode or single-user mode for diagnosing hardware** issues, fixing corrupt file systems, and even reinstalling the operating system. In addition to hardware failures, computer systems can suffer from software errors and poor operating-system performance, which we consider in the following section.

14. PROCESSES

1. Process Concept The process

- 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**.
- The status of the current activity of a process is represented by the value of the program counter and the contents of the processor's registers. The memory layout of a process is typically divided into multiple sections, and is shown in figure 1.21.
- These sections include:
  - **Text section**— the executable code
  - **Data section**—global variables
  - **Heap section**—memory that is dynamically allocated during program runtime
  - **Stack section**— temporary data storage when invoking functions (such as function parameters, return addresses, and local variables)

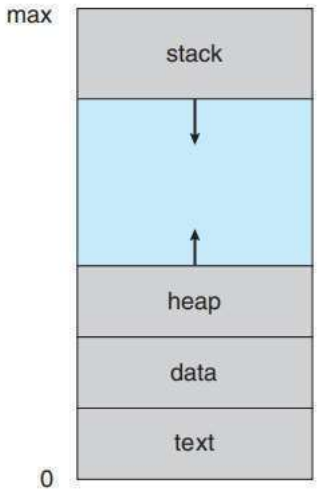


Figure 1.21 - Layout of a process in memory

- Notice that the sizes of the text and data sections are fixed, as their sizes do not change during program run time. However, the stack and heap sections can shrink and grow dynamically during program execution.
- Each time a function is called, an **activation record** containing function parameters, local variables, and the return address is pushed onto the stack; when control is returned from the function, the activation record is popped from the stack.
- Similarly, the heap will grow as memory is dynamically allocated, and will shrink when memory is returned to the system.
- Although the stack and heap sections grow **toward** one another, the operating system must ensure they do not overlap one another.
- A program by itself is **not a process**. A program is a **passive entity,** such as a file containing a list of instructions stored on disk often called an **executable file**).
- A process is an **active entity,** with a program counter specifying the next instruction to execute and a set of associated resources. A program becomes a process when an executable file is loaded into memory. T

Reference Video	
	**Structure of Process** https://youtu.be/grriYn6v76g

2. Process State

As a process executes, it changes **state**. A process may be in one of the following states:

- **New.** - The process is being created.
- **Running.** - Instructions are being executed.
- **Waiting.** - The process is waiting for some event to occur (such as an I/O completion or reception of a signal).
- **Ready.** - The process is waiting to be assigned to a processor.
- **Terminated.** - The process has finished execution.

The state diagram corresponding to these states is shown in below Figure 1.22.

3. 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 1.23. It contains many pieces of information associated with a specific process, including these:

- **Process state**. The state may be new, ready, running, waiting, halted, and so on.
- **Program counter**. The counter indicates the address of the next instruction to be executed for this process.

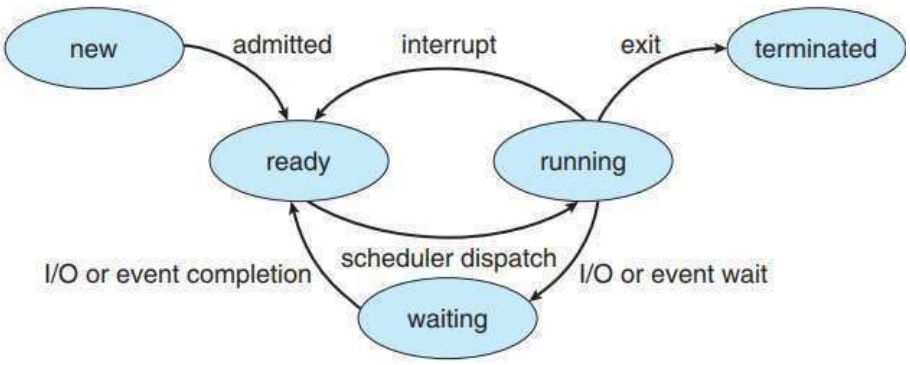


Figure 1.22 - - Diagram of Process state

- **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 informa_x005F_x0002_tion. Along with the program counter, this state information must be saved when an interrupt occurs, to allow the process to be continued correctly afterward when it is rescheduled to run.

- **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

- **Accounting information.** This information includes the amount of CPU and real time used, time limits, account numbers, job or process numbers, and so on.

- **I/O status information**. This information includes the list of I/O devices allocated to the process, a list of open files, and so on.

The PCB simply serves as the **repository for all the data** needed to start, or restart, a process, along with some accounting data.

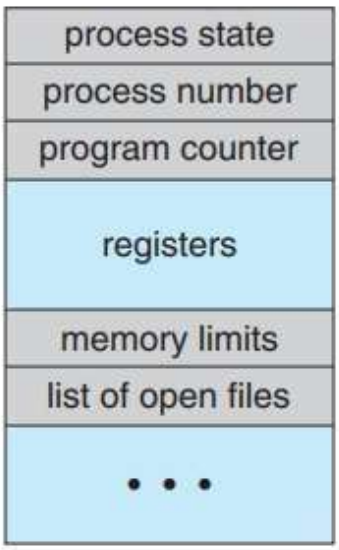


Figure 1.23 - Process Control Block (PCB)

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***.

To meet these objectives, the **process scheduler** selects an available process (possibly from a set of several available processes) for program execution on a core.

Each CPU core can run one process at a time. For a system with a single CPU core, there will **never be more than one process running at a time,** whereas a multicore system can run multiple processes at one time.

If there are more processes than cores, **excess processes will have to wait until a core is free and can be rescheduled**.

The number of processes currently in memory is known as the **degree of multiprogramming.**

- Most processes can be described as either I/O bound or CPU bound.

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

A **CPU-bound process**, in contrast, generates I/O requests infrequently, using more of its time doing computations.

Scheduling Queues

- As processes enter the system, they are put into a **ready queue**, where they are ready and waiting to execute on a CPU's core.
- This queue is generally stored as a linked list; a ready-queue header contains pointers to the first PCB in the list, and each PCB includes a pointer field that points to the next PCB in the ready queue.
- TThe system also includes **other queues**.
- When a process is allocated a CPU core, it executes for a while and eventually terminates, 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 device such as a disk. Since devices run significantly slower than processors, the process will have to wait for the I/O to become available. Processes that are waiting for a certain event to occur — such as completion of I/O — are placed in a **wait queue. (figure 1.24).**

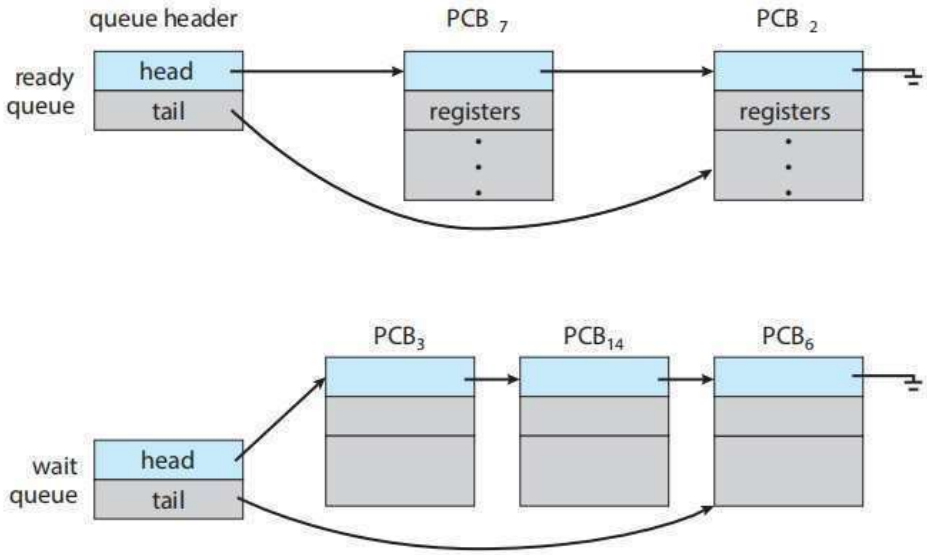


Figure 1.24 - The ready queue and wait queue

A common representation of process scheduling is a **queueing diagram,** such as that in Figure 1.25. Two types of queues are present: the **ready queue and a set of wait 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 a CPU core 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 wait queue.
- The process could create a new child process and then be placed in a wait queue while it awaits the child's termination.
- The process could be removed forcibly from the core, as a result of an
- interrupt or having its time slice expire, and be put back in the ready queue.

In the first two cases, the process eventually **switches from the waiting state to the ready state and is then put back in the ready queue**. A process continues this cycle until it terminates, at which time it is removed from all queues and has its PCB and resources deallocated.

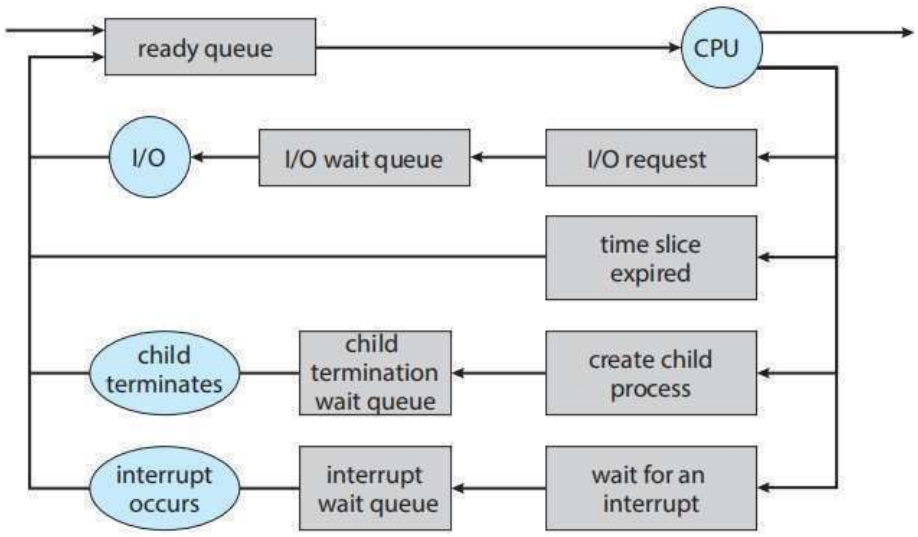


Figure 1.25 - Queueing-diagram representation of process scheduling

CPU Scheduling

- A **process migrates** among the ready queue and various wait queues throughout its lifetime.
- The role of the **CPU scheduler** is to select from among the processes that are in the **ready queue and allocate a CPU core to one of them.** The CPU scheduler must select a new process for the CPU frequently.
- An **I/O-bound process** may execute for only a few milliseconds before waiting for an I/O request.
- Although a **CPU-bound process** will require a CPU core for longer durations, the scheduler is unlikely to grant the core to a process for an extended period. Instead, it is likely designed to forcibly remove the CPU from a process and schedule another process to run.
- Therefore, the CPU scheduler executes at least once every 100 milliseconds, although typically much more frequently.
- Some operating systems have an intermediate form of scheduling, known as **swapping**, whose key idea 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.
- Later, the process can be reintroduced into memory, and its execution can be continued where it left off. This scheme is known as swapping because a process can be "**swapped out"** from memory to disk, where its current status is saved, and later "swapped in" from disk back to memory, where its status is restored.
- Swapping is typically only necessary when memory has been overcommitted and must be freed

Context Switch

- When an interrupt occurs, the system needs to save the current context of the process running on the CPU core 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. It includes the value of the CPU registers, the process state, and memory-management information.
- Switching the CPU core 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 switch (**illustrated in Figure 1.25)
- When a context switch occurs, the kernel saves the context of the old process in its PCB and loads the saved context of the new process scheduled to run.

Context_x005F_x0002_

- switch time is **pure overhead**, because the system does no useful work while switching. Switching speed varies from machine to machine, depending on the memory speed, the number of registers that must be copied, and the existence of special instructions (such as a single instruction to load or store all registers).
- A typical speed is a **several microseconds**.
- Context-switch times are highly dependent on **hardware support.**

Reference Video	
	Context Switching https://youtu.be/w_YCKF323ns

16. OPERATIONS ON PROCESSES

- The processes in most systems can execute concurrently, and they may be created and deleted dynamically.

1. Process Creation

- A process may create several new processes. The creating process is called a **parent process**, and the new processes are called the **children of that process.**

- Each of these new processes may in turn create other processes, forming a tree of processes.

- Most operating systems (including UNIX, Linux, and Windows) identify processes according to a **unique process identifier (or pid)**, which is typically an integer number.

- The pid provides a unique value for each process in the system, and it can be used as an index to access various attributes of a process within the kernel.

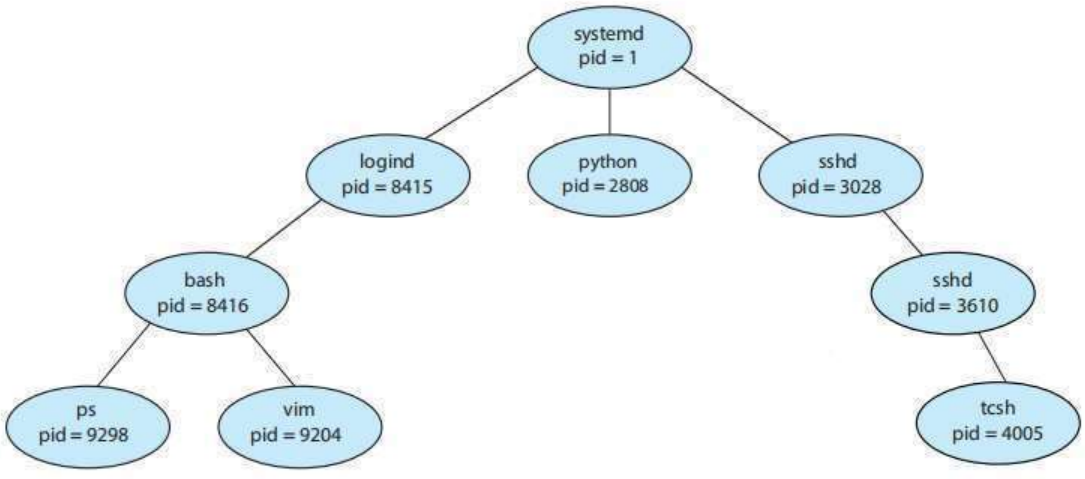


Figure 1.26 - A tree of processes on a typical Linux system

- The figure 1.26 illustrates a typical process tree for the Linux operating system showing the name of each process and its pid. The init process (which always has a pid of 1) serves as the root parent process for all user processes.
- Once the system has booted, the init process can also create various other user processes. The children of init are kthreadd and sshd.

In general, when a process creates a child process, that **child process will need certain resources (CPU time, memory, files, I/O devices) to accomplish its task.**

A child process may be able to obtain its resources **directly from the operating system,** or it may be constrained to a **subset of the resources of the parent process.** The parent may have to partition its resources among its children, or it may be able to share some resources (such as memory or files) among several of its children. When a process creates a new process, two possibilities for execution exist:

1. The parent continues to execute concurrently with its children.
2. The parent waits until some or all of its children have terminated.

There are also two address-space possibilities for the new process:

1. The child process is a duplicate of the parent process (it has the same program and data as the parent).
2. The child process has a new program loaded into it.

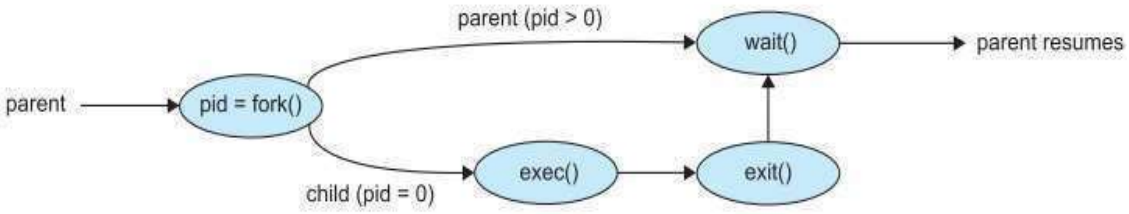


Figure 1.27 - Pocess creation using fork() system call

The parent waits for the child process to complete with the wait() system call. When the child process completes, the parent process resumes from the call to wait(), where it completes using the exit() system call. This is also illustrated in the Figure1.27.

Process Termination

A process terminates when it **finishes executing** its final statement and asks the operating system to delete it by using the exit() system call.

At that point, the process may **return a status value** (typically an integer) to its parent process (via the wait() system call).

All the resources of the process—including physical and virtual memory, open files, and I/O buffers are deallocated by the operating system.

A **parent may terminate the execution of one of its children for a variety of reasons,** such as these:

- The child has **exceeded** its usage of some of the resources that it has been allocated.
- The task assigned to the child is **no longer required**.
- The **parent is exiting**, and the operating system does not allow a child to continue if its parent terminates.

Some systems **do not allow a child to exist if its parent has terminated**. In such systems, if a process terminates (either normally or abnormally), then all its children must also be terminated. This phenomenon, referred to as **cascading termination**, is normally initiated by the operating system.

To illustrate process execution and termination, consider that, in Linux and UNIX systems, we can terminate a process by using the exit() system call, providing an exit status as a parameter:

```
/* exit with status 1 */
exit(1);
```

- When a process terminates, its resources are deallocated by the operating system. However, its entry in the process table must remain there until the parent calls wait(), because the process table contains the process's exit status.
- A process that has terminated, but whose parent has not yet called wait(), is known as a **zombie process**.
- If a parent did not invoke wait() and instead terminated, thereby leaving its child processes as **orphans.**
- Traditional UNIX systems addressed this scenario by assigning the **init process** as the **new parent to orphan processes.**
- The init process periodically invokes wait(), thereby allowing the exit status of any orphaned process to be collected and releasing the orphan's process identifier and process-table entry.

Android Process Hierarchy

Because of resource constraints such as limited memory, mobile operating systems may **have to terminate existing processes to reclaim limited system resources.** Rather than terminating an arbitrary process, Android has identified an importance hierarchy of processes, and when the system must terminate a process to make resources available for a new, or more important, process, it terminates processes in order of increasing importance. From most to least important, **the hierarchy of process classifications** is as follows:

- **Foreground process**—The current process visible on the screen, represent_x005F_x0002_ing

the application the user is currently interacting with

- **Visible process**—A process that is not directly visible on the foreground but that is performing an activity that the foreground process is referring to (that is, a process performing an activity whose status is displayed on the foreground process)

- **Service process**—A process that is similar to a background process but is performing an activity that is apparent to the user (such as streaming music)

- **Background process**—A process that may be performing an activity but is not apparent to the user.

- **Empty process—**A process that holds no active components associated with any application

If system resources must be reclaimed, Android will first terminate empty processes, followed by background processes, and so forth.

INTERPROCESS COMMUNICATION

Processes executing concurrently in the operating system may be either independent processes or cooperating processes. A process is independent if it cannot affect or be affected by the other processes executing in the system. Any process that does not share data with any other process is independent. A process is **cooperating** if it can affect or be affected by the other processes executing in the system. Clearly, any process that shares data with other processes is a cooperating process.

Several reasons for process cooperation:

- **Information sharing**. Since several users may be interested in the same piece of information (for instance, a shared file), we must provide an environment to allow concurrent access to such information.

- **Computation speedup**. If we want a particular task to run faster, we must break it into subtasks, each of which will be executing in parallel with the others.

- **Modularity**. We may want to construct the system in a modular fashion, dividing the system functions into separate processes or threads

- **Convenience**. Even an individual user may work on many tasks at the same time. For instance, a user may be editing, listening to music, and compiling in parallel.

Cooperating processes require an interprocess communication (IPC) mechanism that will allow them to exchange data and information. There are two fundamental models of interprocess communication: **shared memory and message passing**. In the **shared-memory model**, a region of memory that is shared by cooperating processes is established. Processes can then exchange information by reading and writing data to the shared region. In the **message-passing model**, communication takes place by means of messages exchanged between the cooperating processes.

The two communications models are shown in the below figure 1.28

Shared memory can be faster than message passing, since message-passing systems are typically implemented using system calls and thus require the more time-consuming task of kernel intervention. In shared-memory systems, system calls are required only to establish shared memory regions. Once shared memory is established, all accesses are treated as routine memory accesses, and no assistance from the kernel is required.

IPC in Shared-Memory Systems

- Inter process communication using shared memory requires communicating processes to establish a region of shared memory. They can then exchange information by reading and writing data in the shared areas.
- Consider the **producer–consumer problem,** which is a common paradigm for cooperating processes. A **producer** process produces information that is consumed by a **consumer** process.
- One solution to the **producer–consumer** problem uses shared memory. To allow producer and consumer processes to run concurrently, we must have available a buffer of items that can be filled by the producer and emptied by the consumer. This buffer will reside in a region of memory that is shared by the producer and consumer processes.
- A producer can produce one item while the consumer is consuming another item. The producer and consumer must be synchronized, so that the consumer does not try to consume an item that has not yet been produced.

Two types of buffers can be used. The unbounded buffer places no practical limit on the size of the buffer. The consumer may have to wait for new items, but the producer can always produce new items. The bounded buffer assumes a fixed buffer size. In this case, the consumer must wait if the buffer is empty, and the producer must wait if the buffer is full. The following variables reside in a region of memory shared by the producer and consumer processes.

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

The shared buffer is implemented as a circular array with two logical pointers: in and out. The variable in points to the next free position in the buffer; out points to the first full position in the buffer. The buffer is empty when in == out; the buffer is full when ((in + 1) % BUFFER SIZE) == out.

```
item next produced;
while (true) {
/* produce an item in next produced */
while (((in + 1) % BUFFER SIZE) == out)
; /* do nothing */
buffer[in] = next produced;
in = (in + 1) % BUFFER SIZE;
}
```

The code for the producer process is shown in the above Figure, and the code for the consumer process is shown in the below Figure . The producer process has a local variable next produced in which the new item to be produced is stored. The consumer process has a local variable next consumed in which the item to be consumed is stored.

```
item next consumed;

while (true) {

while (in == out)

; /* do nothing */

next consumed = buffer[out];

out = (out + 1) % BUFFER SIZE;

/* consume the item in next consumed */
```

IPC in Message-Passing Systems

Here cooperating processes communicate with each other via a message-passing facility. Message passing provides a mechanism to allow processes to communicate and to synchronize their actions without sharing the same address space . . A message-passing facility provides at least two operations:

send(message) **receive(message)**

- Messages sent by a process can be either fixed or variable in size. If processes P and Q want to communicate, they must send messages to and receive messages from each other: a communication link must exist between them. This link can be implemented in a variety of ways. We are concerned here not with the link's physical implementation but rather with its logical implementation.

Here are several methods for logically implementing a link and the send()/receive() operations:

- • Direct or indirect communication
- • Synchronous or asynchronous communication
- • Automatic or explicit buffering

Naming

- Processes that want to communicate must have a way to refer to each other. They can use either direct or indirect communication. Under direct communication, each process that wants to communicate must explicitly name the recipient or sender of the communication. In this scheme, the send() and receive() primitives are defined as:
- send(P, message)—Send a message to process P.
- receive(Q, message)—Receive a message from process Q.

A communication link in this scheme has the following properties:

- A link is established automatically between every pair of processes that want to communicate. The processes need to know only each other's identity to communicate.
- A link is associated with exactly two processes.
- Between each pair of processes, there exists exactly one link.
- This scheme exhibits symmetry in addressing; that is, both the sender process and the receiver process must name the other to communicate. A variant of this scheme employs asymmetry in addressing. Here, only the sender names the recipient; the recipient is not required to name the sender. In this scheme, the send() and receive() primitives are defined as follows:

- send(P, message)—Send a message to process P.
- receive(id, message)—Receive a message from any process. The variable id is set to the name of the process with which communication has taken place.

- With indirect communication, the messages are sent to and received from mailboxes, or ports. A mailbox can be viewed abstractly as an object into which messages can be placed by processes and from which messages can be removed. Each mailbox has a unique identification. A process can communicate with another process via a number of different mailboxes, but two processes can communicate only if they have a shared mailbox.

- The send() and receive() primitives are defined as follows:
  - send(A, message)—Send a message to mailbox A.
  - receive(A, message)—Receive a message from mailbox A.

- In this scheme, a communication link has the following properties:
  - A link is established between a pair of processes only if both members of the pair have a shared mailbox.
  - A link may be associated with more than two processes.

- Between each pair of communicating processes, a number of different links may exist, with each link corresponding to one mailbox.

- Now suppose that processes P1, P2, and P3 all share mailbox A. Process P1 sends a message to A, while both P2 and P3 execute a receive() from A. Which process will receive the message sent by P1? The answer depends on which of the following methods we choose:

- Allow a link to be associated with two processes at most.
- Allow at most one process at a time to execute a receive() operation.
- Allow the system to select arbitrarily which process will receive the message (that is, either P2 or P3, but not both, will receive the message). The system may define an algorithm for selecting which process will receive the message

Synchronization

- Communication between processes takes place through calls to send() and receive() primitives. There are different design options for implementing each primitive. Message passing may be either blocking or nonblocking— also known as synchronous and asynchronous.
- **Blocking send**. The sending process is blocked until the message is received by the receiving process or by the mailbox.
- **Nonblocking send.** The sending process sends the message and resumes operation.
- **Blocking receive**. The receiver blocks until a message is available.
- Nonblocking receive. The receiver retrieves either a valid message or a null.

Buffering

- Whether communication is direct or indirect, messages exchanged by communicating processes reside in a temporary queue. Basically, such queues can be implemented in three ways:
- **Zero capacity.** The queue has a maximum length of zero; thus, the link cannot have any messages waiting in it. In this case, the sender must block until the recipient receives the message.
- **Bounded capacity**. The queue has finite length n; thus, at most n messages can reside in it. If the queue is not full when a new message is sent, the message is placed in the queue, and the sender can continue execution without waiting. The link's capacity is finite, however. If the link is full, the sender must block until space is available in the queue.
- **Unbounded capacity**. The queue's length is potentially infinite; thus, any number of messages can wait in it. The sender never blocks.
- The **zero-capacity** case is sometimes referred to as a message system with no buffering. The other cases are referred to as systems with automatic buffering.

Lecture Slides

R.M.K
GROUP OF
INSTITUTIONS

Lecture Slides

Lecture Slides
https://drive.google.com/drive/folders/1Iy8yFXuDs_p03Tm8G0tc1l1v21PYe7Ic?usp=sharing

Lecture Videos

Lecture Videos

Lecture Videos
https://drive.google.com/drive/folders/1qZLfekm_yUusgA69eEElh9WxZBI1juh9?usp=sharing

R.M.K GROUP OF INSTITUTIONS

Assignment

R.M.K
GROUP OF
INSTITUTIONS

Assignment

S.No	Question	Category
1.	a. What is the main difculty that a programmer must overcome in writing an operating system for a real-time environment? b. Distinguish between the client –server and peer-to-peer models of distributed systems.	5
2.	a. What are the three main purposes of an operating system? b. What is the purpose of system programs? c. What is the purpose of system calls?	4

3.	a. Including the initial parent process, how many processes are created by the program shown in Figure
	`#include <stdio.h>`
	`#include <unistd.h>`
	`int main()`
	`{`
	`/* fork a child process */`
	`fork();`
	`/* fork another child process */`
	`fork();`
	`/* and fork another */`
	`fork();`
	`return 0;`
	`}`
	b. Give two reasons why caches are useful. What problems do they solve? What problems do they cause? If a cache can be made as large as the device for which it is caching (for instance, a cache as large as a disk), why not make it that large and eliminate the device?
		3

Assignment

S.No	Question	Category
4.	a. What system calls have to be executed by a command interpreter or shell in order to start a new process on a UNIX system? b. List five services provided by an operating system, and explain how each creates convenience for users. In which cases would it be impossible for user-level programs to provide these services? Explain your answer.	2
5.	Using the program shown in Figure 3.30, explain what the output will be at LINE A. b. Some computer systems provide multiple register sets. Describe what happens when a context switch occurs if the new context is already loaded into one of the register sets. What happens if the new context is in memory rather than in a register set and all the register sets are in use?	1

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

int main()
{
    /* fork a child process */
    fork();

    /* fork another child process */
    fork();

    /* and fork another */
    fork();

    return 0;
}
```

Part A Q & A

PART -A

1. Define operating system. (CO1,K2)

An ***operating system*** acts as an intermediary between the user of a computer and the computer hardware.

Goal:

The purpose of an operating system is to provide an environment in which a user can execute programs in a ***convenient*** and ***efficient*** manner.

2.List the components or structural elements of the computer system. (CO1,K1)

There are four main structural elements:

Processor: Controls the operation of the computer and performs its data processing functions.

Main memory: Stores data and programs. This memory is typically volatile

I/O modules: Move data between the computer and its external environment.

System bus: Provides for communication among processors, main memory, and I/O modules.

3. List the steps in instruction execution. (CO1,K1)

Instruction processing consists of two steps. The two steps are referred to as the ***fetch stage*** and the ***execute stage***.:

The processor reads (fetches) instructions from memory one at a time and executes each instruction.

Program execution consists of repeating the process of instruction fetch and instruction execution. The processing required for a single instruction is called an ***instruction cycle***.

4. What is an interrupt? (CO1,K2)

The sequence of instructions which suspends the normal execution of the program is called as interrupt. Interrupts are provided primarily as a way to improve processor utilization

PART -A

5) Define Cache Memory.(CO1,K2)

Cache memory is the fastest memory that contains a copy of a portion of main memory. Cache memory is intended to provide memory access time approaching that of the fastest memories available.

6) Define Locality of Reference. (CO1,K2)

When a block of data is fetched into the cache to satisfy a single memory reference, it is likely that many of the near-future memory references will be to other bytes in the block. This phenomenon is called as locality of reference.

7) Define DMA. (CO1,K2)

When large volumes of data are to be moved, a more efficient technique is required: direct memory access (DMA). The DMA module transfers the entire block of data, one word at a time, directly to or from memory without going through the processor. When the transfer is complete, the DMA module sends an interrupt signal to the processor.

8) List the objectives and functions of OS. (CO1,K1)

The objectives of OS are:

Convenience: An OS makes a computer more convenient to **use**.

Efficiency: An OS allows the **computer** system resources to be used in an efficient manner.

Ability to evolve: An OS should be constructed in such a way as to permit the effective development, testing, and introduction of new system functions without interfering with service.

9) List the Stages of Operating system. (CO1,K1)

Operating Systems Stages include Serial Processing

Simple Batch Systems

PART -A

10) What is a Trap? (CO1,K2)

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.

11) What is a system call?(CO1,K2)

A system call is a routine which acts as an interface between the user mode and the kernel mode.

12) Define bootstrap program.(CO1,K2)

The procedure of starting a computer by loading the kernel is known as booting the system. On most computer systems, a small piece of code known as the bootstrap program or bootstrap loader locates the kernel, loads it into main memory, and starts its execution.

13) What is an interrupt vector?(CO1,K2)

The locations that hold the addresses of the interrupt service routines for the various devices is called as **interrupt vector**,

14) Define uniprogramming vs multiprogramming.(CO1,K2)

In **uniprogramming**, the processor spends certain amount of time executing, until it reaches an I/O instruction. It must then wait until that I/O instruction concludes before proceeding.

In Multiprogramming when one job needs to wait for I/O, the processor can switch to the other job.

15) Do timesharing differs from multiprogramming? If so, How?(CO1,K2)

Multiprogramming allows using the CPU effectively by allowing various users to use the CPU and I/O devices effectively. Multiprogramming makes sure that the CPU always has something to execute, thus increases the CPU utilization. On the other hand, Time sharing is the sharing of computing resources among several users at the same time. 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.

PART -A

16) Why API's need to be used rather than system calls? (CO1,K2)

System calls differ from platform to platform. By using a API, it is easier to migrate your software to different platforms.

The API usually provides more useful functionality than the system call directly. For example the 'fork' API includes tons of code beyond just making the 'fork' system call. So does 'select'.

The API can support multiple versions of the operating system and detect which version it needs to use at run time.

17) What is the purpose of system programs?(CO1,K2)

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

18) How does an interrupt differ from a trap?(CO1,K2)

A trap is a software-generated interrupt. An interrupt can be used to signal the completion of an I/O to obviate the need for device polling. A trap can be used to call operating system routines or to catch arithmetic errorsInterrupts are hardware interrupts, while traps are software-invoked interrupts.

19. What is multicore Programming? (K1)(CO2)

Multiple computing cores on a single chip is called multicore or multiprocessor systems.

20. Define Process. (K1)(CO2)

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.

21. List the states of the process.(K1)(CO2)

As a process executes, it changes state. A process may be in one of the following states:

New: The process is being created.

Running: Instructions are being executed.

Waiting: The process is waiting for some event to occur

Ready: The process is waiting to be assigned to a processor.

Terminated: The process has finished execution.

22. . What is Process Control Block (PCB)?(K1,CO2)

Each process is represented in the operating system by a process control block (PCB)—also called a task control block. A PCB includes the following:

Process state

Program counter

CPU registers

CPU-scheduling information

Memory-management information

Accounting information.

I/O status information

23. Define Thread.(K1,CO2)

A thread is a light weight process involved in the execution. A thread is a basic unit of CPU utilization; it comprises a thread ID, a program counter, a register set, and a stack. Single thread of control allows the process to perform only one task at a time.

24. Define independent vs Co-operative Process? (K1,CO2)

A process is independent if it cannot affect or be affected by the other processes executing in the system. Any process that does not share data with any other process is independent. A process is cooperating if it can affect or be affected by the other processes executing in the system. Clearly, any process that shares data with other processes is a cooperating process.

25. Define IPC. (K1,CO2)

Cooperating processes require an interprocess communication (IPC) mechanism that will allow them to exchange data and information. There are two fundamental models of interprocess communication: shared memory and message passing.

26 .Define the fundamental models of IPC.(K1,CO2)

In the shared-memory model, a region of memory that is shared by cooperating processes is established. In the message-passing model, communication takes place by means of messages exchanged between the cooperating processes.

27. What is Context Switch? (K1,CO2)

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 switch.

28. Define Cascading Termination.(K1,CO2)

Some systems do not allow a child to exist if its parent has terminated. In such systems, if a process terminates, then all its children must also be terminated. This phenomenon, referred to as cascading termination.

29. What is zombie process?(K1,CO2)

When a process terminates, its resources are deallocated by the operating system. However, its entry in the process table must remain there until the parent calls wait(). A process that has terminated, but whose parent has not yet called wait(), is known as a zombie process.

30. Define independent vs Co-operative Process? (K1,CO2)

A process is independent if it cannot affect or be affected by the other processes executing in the system. Any process that does not share data with any other process is independent. A process is cooperating if it can affect or be affected by the other processes executing in the system. Clearly, any process that shares data with other processes is a cooperating process.

31. Define IPC. (K1,CO2)

Cooperating processes require an interprocess communication (IPC) mechanism that will allow them to exchange data and information. There are two fundamental models of interprocess communication: shared memory and message passing.

32. Define the fundamental models of IPC.(K1,CO)

In the shared-memory model, a region of memory that is shared by cooperating processes is established. In the message-passing model, communication takes place by means of messages exchanged between the cooperating processes.

Part B Q

PART -B

1. Define Interrupt. Explain how interrupt is handled by Interrupt handler. (CO1,K2)
2. Explain in detail about Multi-core and Multi processor Architecture. (CO1,K2)
3. Explain structure of OS.
4. Explain dual mode operation of OS
5. Explain the structure and operations of OS.
6. Explain the steps involved in instruction execution
7. Draw the state diagram of a process from- its creation to termination, including all transitions, and briefly illustrates every state and every transition. (CO1,K3)
8. 5Explain in detail about Inter Process Communication with an example? (13) (CO1,K3)
9. Explain about the shared memory model and Message passing system? (13) (CO1,K2)
10. Explain Virtualization in detail. (CO1,K3)
11. Discuss in detail about various system calls provided by OS. (CO1,K2)
12. Justify the responsibilities of OS for managing various resources (CO1,K3)
13. Explain symmetric and asymmetric multiprocessing. Compare the differences between symmetric and asymmetric multiprocessing .What are three advantages and one disadvantage of multiprocessor systems? (CO1,K2)

Supportive Online Certification courses

R.M.K GROUP OF INSTITUTIONS

SUPPORTIVE ONLINE COURSES

S No	Course provider	Course title	Link
1	Udemy	Operating Systems from scratch - Part 1	https://www.udemy.com/course/operating-systems-from-scratch-part1/
2	Udacity	Introduction to Operating Systems	https://www.udacity.com/course/introduction-to-operating-systems
3	Coursera	Operating Systems and You: Becoming a Power User	https://www.coursera.org/learn/os-power-user
4	edX	Computer Hardware and Operating Systems	https://www.edx.org/course/computer-hardware-and-operating-systems

Real life Applications in day to day life and to Industry

R.M.K GROUP OF INSTITUTIONS

REAL TIME APPLICATIONS IN DAY TO DAY LIFE AND TO INDUSTRY

1. Explain about the Mobile Operating Systems. (K4, CO1)
2. Operating systems used in various companies. (K4, CO1)
3. Need of an own operating systems for an organization. (K4, CO1)

Content beyond Syllabus

Contents beyond the Syllabus

Distributed operating system

Reference Video – Content Beyond Syllabus

https://www.youtube.com/watch?v=NYBKXzl5bW U

Assessment Schedule

ASSESSMENT SCHEDULE

FIAT

Proposed date : 14.02.2024

R.M.K GROUP OF INSTITUTIONS

Prescribed Text books & Reference books

R.M.K GROUP OF INSTITUTIONS

PRESCRIBED TEXT BOOKS AND REFERENCE BOOKS

TEXT BOOKS

✓ Silberschatz Abraham, Greg Gagne, Peter B. Galvin. “Operating System Concepts”, Tenth Edition, Wiley, 2018. [EBOOK]

REFERENCE BOOKS

✓ 1.William Stallings, Operating Systems – Internals and Design Principles, Pearson Education, New Delhi, 2018.

✓ 2.Achyut S.Godbole, Atul Kahate, Operating Systems||, McGraw Hill Education, 2016.

✓ Andrew S. Tanenbaum, "Modern Operating System", 4 th Edition, PHI Learning, New Delhi, 2018.

Mini Project Suggestions

R.M.K GROUP OF INSTITUTIONS

MINI PROJECT SUGGESTIONS

Category	Mini Project Title
1	**Message Passing System:** Develop a messaging system that enables communication between multiple processes using IPC mechanisms
	Multi-threaded Web Crawler: Implement a multi-threaded web crawler. The crawler should be able to remember the last URLs and able to resume. Your program should be able to create appropriate number of threads
2	**Shared Memory Chat Application:** Create a chat application where processes communicate through shared memory for message exchange.
	Process Manager: Identify the system and user processes. For each process provide CPU, memory, and I/O utilization. You should also tag a process as a CPU bound or I/O bound.
3	Build a C program that Virtualizing a Memory.
	Custom Shell Prompt: Design a customized shell prompt with dynamic information display, colors, and user-specific features to enhance the user interface.
4	Construct a Animation Video That Shows the Different States of Process
	Create a Report on Different Operating System Tools used to Perform Various functions.
5	**Custom System Call Implementation:** Implement a custom system call and demonstrate its usage in a simple user-space program.
	Introduction Module: Include an interactive introduction module that explains the importance of operating systems in everyday computing.

Thank you

Disclaimer: