## DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

# *COURSE MATERIALS*

# *CST 206 OPERATING SYSTEMS*



### VISION OF THE INSTITUTION

To mould true citizens who are millennium leaders and catalysts of change through excellence in education.

### MISSION OF THE INSTITUTION

**NCERC** is committed to transform itself into a center of excellence in Learning and Research in Engineering and Frontier Technology and to impart quality education to mould technically competent citizens with moral integrity, social commitment and ethical values.

We intend to facilitate our students to assimilate the latest technological know-how and to imbibe discipline, culture and spiritually, and to mould them in to technological giants, dedicated research scientists and intellectual leaders of the country who can spread the beams of light and happiness among the poor and the underprivileged.

## ABOUT DEPARTMENT

- Established in: 2002
- Course offered : B.Tech in Computer Science and Engineering

  M.Tech in Computer Science and Engineering

  M.Tech in Cyber Security
- Approved by AICTE New Delhi and Accredited by NAAC
- Affiliated to A P J Abdul Kalam Technological University.

## DEPARTMENT VISION

Producing Highly Competent, Innovative and Ethical Computer Science and Engineering Professionals to facilitate continuous technological advancement.

## DEPARTMENT MISSION

1. To Impart Quality Education by creative Teaching Learning Process
2. To Promote cutting-edge Research and Development Process to solve real world problems with emerging technologies.
3. To Inculcate Entrepreneurship Skills among Students.
4. To cultivate Moral and Ethical Values in their Profession.
5.

### PROGRAMME EDUCATIONAL OBJECTIVES

**PEO1:** Graduates will be able to Work and Contribute in the domains of Computer Science and Engineering through lifelong learning.

**PEO2:** Graduates will be able to Analyse, design and development of novel Software Packages, Web Services, System Tools and Components as per needs and specifications.

**PEO3:** Graduates will be able to demonstrate their ability to adapt to a rapidly changing environment by learning and applying new technologies.

**PEO4:** Graduates will be able to adopt ethical attitudes, exhibit effective communication skills, Teamworkand leadership qualities.

## PROGRAM OUTCOMES (POS)

**Engineering Graduates will be able to:**

1. **Engineering knowledge**: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.
2. **Problem analysis**: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
3. **Design/development of solutions**: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.
4. **Conduct investigations of complex problems**: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
5. **Modern tool usage**: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.
6. **The engineer and society**: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
7. **Environment and sustainability**: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
8. **Ethics**: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
9. **Individual and team work**: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
10. **Communication**: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
11. **Project management and finance**: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
12. **Life-long learning**: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

## PROGRAM SPECIFIC OUTCOMES (PSO)

**PSO1**: Ability to Formulate and Simulate Innovative Ideas to provide software solutions for Real-time Problems and to investigate for its future scope.

**PSO2**: Ability to learn and apply various methodologies for facilitating development of high quality System Software Tools and Efficient Web Design Models with a focus on performance

optimization.

**PSO3**: Ability to inculcate the Knowledge for developing Codes and integrating hardware/software products in the domains of Big Data Analytics, Web Applications and Mobile Apps to create innovative career path and for the socially relevant issues.

## COURSE OUTCOMES

**Prerequisite:** Topics covered in the courses are **Data Structures (CST 201)** and **Programming in C (EST 102)**

**Course Outcomes**: After the completion of the course the student will be able to

| | |
|---|---|
| CO1 | Explain the relevance, structure and functions of Operating Systems in computing devices. (**Cognitive knowledge: Understand**) |
| CO2 | Illustrate the concepts of process management and process scheduling mechanisms employed in Operating Systems. (**Cognitive knowledge: Understand**) |
| CO3 | Explain process synchronization in Operating Systems and illustrate process synchronization mechanisms using Mutex Locks, Semaphores and Monitors (**Cognitive knowledge: Understand**) |
| CO4 | Explain any one method for detection, prevention, avoidance and recovery for managing deadlocks in Operating Systems. (**Cognitive knowledge: Understand**) |
| CO5 | Explain the memory management algorithms in Operating Systems. (**Cognitive knowledge: Understand**) |
| CO6 | Explain the security aspects and algorithms for file and storage management in Operating Systems. **(Cognitive knowledge: Understand)** |

## MAPPING OF COURSE OUTCOMES WITH PROGRAM OUTCOMES

| CO'S | PO1 | PO2 | PO3 | PO4 | PO5 | PO6 | PO7 | PO8 | PO9 | PO10 | PO11 | PO12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| C212.1 | 3 | 2 | 2 | 2 | | | | | | | | 3 |
| C212.2 | 3 | 3 | 2 | 2 | | | | | | | | 3 |
| C212.3 | 3 | 3 | 2 | 2 | 2 | | | | | | | 3 |
| C212.4 | 3 | 3 | 2 | 2 | | | | | | | | 3 |
| C212.5 | 3 | 3 | 3 | 3 | 2 | | | 2 | | | | 3 |
| C212 | 3 | 2.8 | 2.2 | 2.2 | 2 | - | - | 2 | - | - | - | 3 |

**Note: H-Highly correlated=3, M-Medium correlated=2, L-Less correlated=1**

| CO'S | PSO1 | PSO2 | PSO3 |
|---|---|---|---|
| C212.1 | 3 | 2 | |
| C212.2 | 3 | 3 | 2 |
| C212.3 | 3 | 2 | 2 |
| C212.4 | 3 | 3 | 2 |
| C212.5 | 3 | 3 | 3 |
| C311 | 3 | 2.6 | 2.25 |

## SYLLABUS

**Syllabus**

**Module I**

**Introduction:** Operating system overview – Operations, Functions, Service – System calls, Types – Operating System structure - Simple structure, Layered approach, Microkernel, Modules – System boot process.

**Module II**

**Processes** - Process states, Process control block, threads, scheduling, Operations on processes - process creation and termination – Inter-process communication - shared memory systems, Message passing systems.

**Process Scheduling** – Basic concepts- Scheduling criteria -scheduling algorithms- First come First Served, Shortest Job Firs, Priority scheduling, Round robin scheduling

**Module III**

**Process synchronization**- Race conditions – Critical section problem – Peterson's solution, Synchronization hardware, Mutex Locks, Semaphores, Monitors – Synchronization problems - Producer Consumer, Dining Philosophers and Readers-Writers.

**Deadlocks:** Necessary conditions, Resource allocation graphs, Deadlock prevention, Deadlock avoidance – Banker's algorithms, Deadlock detection, Recovery from deadlock.

**Module IV**

**Memory Management:** Concept of address spaces, Swapping, Contiguous memory allocation, fixed and variable partitions, Segmentation, Paging. Virtual memory, Demand paging, Page replacement algorithms.

**Module V**

**File System:** File concept - Attributes, Operations, types, structure – Access methods, Protection. File-system implementation, Directory implementation. Allocation methods.
**Storage Management:** Magnetic disks, Solid-state disks, Disk Structure, Disk scheduling, Disk formatting.

**Text Book**

Abraham Silberschatz, Peter Baer Galvin, Greg Gagne, ' Operating System Concepts' 9th Edition, Wiley India 2015.

**Reference Books:**


1. Andrew S Tanenbaum, "Modern Operating Systems" , 4th Edition, Prentice Hall, 2015.
2. William Stallings, "Operating systems", 6th Edition, Pearson, Global Edition, 2015.
3. Garry Nutt, Nabendu Chaki, Sarmistha Neogy, "Operating Systems", 3rd Edition, Pearson Education.
4. D.M.Dhamdhere, "Operating Systems", 2nd Edition, Tata McGraw Hill, 2011.
5. Sibsankar Haldar, Alex A Aravind, "Operating Systems", Pearson Education.

## QUESTION BANK

| Q:NO: | MODULE I | CO | KL | Page No |
|---|---|---|---|---|
| 1 | Explain module based operating system. | CO1 | K2 | 18 |
| 2 | Explain micro kernel in detail with neat diagram | CO1 | K2 | 17 |
| 3 | Explain operating system implementation strategies | CO1 | K2 | |
| 4 | Explain system call with an example | CO1 | K2 | 11 |
| 5 | Explain implementation of system call in detail | CO1 | K2 | 11 |
| 6 | With figure explain abstract components of operating system. | CO1 | K2 | 1 |
| 7 | Differentiate system view and user view | CO1 | K4 | 3 |
| 8 | What you mean by dual mode operation in operating system, explain in detail | CO1 | K2 | 5 |
| 9 | List out and explain functions of operating system. | CO1 | K2 | 7 |
| 10 | Explain various services of an operating system | CO1 | K2 | 9 |
| 11 | Explain system boot process in detail | CO1 | K2 | 19 |
| | **MODULE II** | | | |
| 1 | With the help of an appropriate diagram explain various process states | CO2 | K2 | 22 |
| 2 | Explain PCB in detail | CO2 | K2 | 23 |
| 3 | What are the advantages of multi threaded programming? Explain | CO2 | K5 | 25 |
| 4 | Differentiate various process scheduling queues | CO2 | K4 | 27 |
| 5 | Differentiate various schedulers available in an operating system | CO2 | K4 | 28 |
| 6 | Write short notes about context switching | CO2 | K2 | 28 |
| 7 | Describe various operations on processes | CO2 | K2 | 29 |

<table>
<tr><td>8</td><td>Differentiate the two models of inter process communication</td><td>CO2</td><td>K4</td><td>32</td></tr>
<tr><td>9</td><td>How you can solve bounded buffer problem using shared memory? Explain in detail</td><td>CO2</td><td>K5</td><td>34</td></tr>
<tr><td>10</td><td>Differentiate preemptive and non preemptive scheduling</td><td>CO2</td><td>K4</td><td>36</td></tr>
<tr><td>11</td><td>What are the various criteria used in scheduling?</td><td>CO2</td><td>K2</td><td>37</td></tr>
<tr><td>12</td><td>find the average waiting time and average turn around time of the process with following scheduling algorithms<br>i) SJFS scheduling(non preemptive) ii) Priority scheduling(preemptive)<br><table><tr><td>Process</td><td>Arrival Time</td><td>Burst Time</td><td>Priority</td></tr><tr><td>P1</td><td>0</td><td>5</td><td>2</td></tr><tr><td>P2</td><td>4</td><td>3</td><td>1</td></tr><tr><td>P3</td><td>2</td><td>9</td><td>4</td></tr><tr><td>P4</td><td>6</td><td>4</td><td>3</td></tr></table></td><td>CO2</td><td>K6</td><td>40</td></tr>
<tr><td>13</td><td>Find the average waiting time and average turn around time of the process with following scheduling algorithms<br>i) Round robin scheduling ii) Priority scheduling(non preemptive)<br><table><tr><td>Process</td><td>Arrival Time</td><td>Burst Time</td><td>Priority</td></tr><tr><td>P1</td><td>0</td><td>6</td><td>1</td></tr><tr><td>P2</td><td>2</td><td>3</td><td>3</td></tr><tr><td>P3</td><td>4</td><td>5</td><td>2</td></tr><tr><td>P4</td><td>5</td><td>2</td><td>4</td></tr></table></td><td>CO2</td><td>K6</td><td>42</td></tr>
<tr><td>14</td><td>find the average waiting time and average turn around time of the process with following scheduling algorithms<br>i) Round robin scheduling ii) FCFS scheduling</td><td>CO2</td><td>K6</td><td>42</td></tr>
</table>

| | | | | |
|---|---|---|---|---|
| | <table><tr><th>Process</th><th>Arrival Time</th><th>Burst Time</th><th>Priority</th></tr><tr><td>P1</td><td>2</td><td>4</td><td>4</td></tr><tr><td>P2</td><td>0</td><td>3</td><td>3</td></tr><tr><td>P3</td><td>3</td><td>6</td><td>1</td></tr><tr><td>P4</td><td>6</td><td>8</td><td>2</td></tr></table> | | | |

## MODULE III

| | | | | |
|---|---|---|---|---|
| 1 | Explain any critical section problem in detail | CO3 | K2 | 46 |
| 2 | Explain the necessary conditions that a system is in deadlock state. | CO3 | K2 | 61 |
| 3 | Describe the term Hold-and-Wait in detail Explain the safety Algorithm in detail. | CO3 | K2 | 66 |
| 4 | Describe Deadlock Avoidance Method in OS | CO3 | K2 | |
| 5 | Explain the concept of deadlock prevention | CO3 | K2 | 64 |
| 6 | Explain the term circular wait in detail. | CO3 | K2 | 65 |
| 7 | Write a short note on Resource Allocation Graph. | CO3 | K2 | 62 |
| 8 | Consider the following snapshot of a system<br><table><tr><th></th><th colspan="3">Allocation</th><th colspan="3">Max</th><th colspan="3">Available</th></tr><tr><th></th><th>A</th><th>B</th><th>C</th><th>A</th><th>B</th><th>C</th><th>A</th><th>B</th><th>C</th></tr><tr><td>$P_0$</td><td>0</td><td>1</td><td>0</td><td>7</td><td>5</td><td>3</td><td>3</td><td>3</td><td>2</td></tr><tr><td>$P_1$</td><td>2</td><td>0</td><td>0</td><td>3</td><td>2</td><td>2</td><td></td><td></td><td></td></tr><tr><td>$P_2$</td><td>3</td><td>0</td><td>2</td><td>9</td><td>0</td><td>2</td><td></td><td></td><td></td></tr><tr><td>$P_3$</td><td>2</td><td>1</td><td>1</td><td>2</td><td>2</td><td>2</td><td></td><td></td><td></td></tr><tr><td>$P_4$</td><td>0</td><td>0</td><td>2</td><td>4</td><td>3</td><td>3</td><td></td><td></td><td></td></tr></table><br>Find the Safe Sequence of processes using banker's Algorithm. | CO3 | K6 | 68 |
| 9 | Describe deadlock detection techniques in detail. | CO3 | K2 | 71 |
| 10 | Write a short note on Bankers Algorithm with Example. | CO3 | K5 | 68 |
| 11 | Explain semaphores with an example | CO3 | K5 | 53 |
| 12 | Explain Peterson's solution | CO3 | K2 | 48 |
| 13 | Describe Deadlock Avoidance Method in OS | CO3 | K2 | 73 |

| | | | | |
|---|---|---|---|---|
| 14 | Explain Dining philosopher's problem | CO3 | K2 | 59 |
| | **MODULE IV** | | | |
| 1 | Differentiate logical and physical address. | CO4 | K4 | 77 |
| 2 | Explain Dynamic loading | CO4 | K2 | 78 |
| 3 | Explain the process of swapping with suitable diagrams | CO4 | K2 | 79 |
| 4 | Differentiate various memory allocation strategies | CO4 | K4 | 81 |
| 5 | Discuss about Address Space in detail. | CO4 | K2 | 76 |
| 6 | Explain about Demand Paging in detail. | CO4 | K2 | 94 |
| 7 | A system uses 3 page frames for storing process pages in main memory. It uses the First in First out (FIFO) page replacement policy. Assume that all the page frames are initially empty. What is the total number of page faults that will occur while processing the page reference string given below-<br>4,3,2,4,5,2,7,3,5,4 | CO4 | K5 | 96 |
| 8 | Explain page table. | CO4 | K2 | 84 |
| 9 | Explain about Segmentation in detail. | CO4 | K2 | 90 |
| 10 | Explain paging in detail | CO4 | K2 | 82 |
| 11 | A system uses 3 page frames for storing process pages in main memory. It uses the LRU page replacement policy. Assume that all the page frames are initially empty. What is the total number of page faults that will occur while processing the page reference string given below-<br>4,3,2,4,5,2,7,3,5,4 | CO4 | K5 | 97 |
| 12 | A system uses 3 page frames for storing process pages in main memory. It uses the Optimal Page Replacement policy. Assume that all the page frames are initially empty. What is the total number of page faults that will occur while processing the page reference string given below- | CO4 | K5 | 98 |

| | | | | |
|---|---|---|---|---|
| | 4,3,2,4,5,2,7,3,5,4 | | | |
| | **MODULE V** | | | |
| 1 | Explain about File Access Methods | CO5 | K2 | 111 |
| 2 | Describe about Files in OS. | CO5 | K2 | 109 |
| 3 | Explain in detail about File Types. | CO5 | K2 | 110 |
| 4 | Describe File Structures in detail. | CO5 | K2 | 111 |
| 5 | Explain in detail about file operations. | CO5 | K2 | 109 |
| 6 | Explain File Attributes in OS. | CO5 | K2 | 109 |
| 7 | Explain storage management in detail | CO5 | K2 | 101 |
| 8 | Differentiate various approaches of disk scheduling | CO5 | K4 | 103 |
| 9 | Explain disk formatting in detail | CO5 | K2 | 108 |
| 10 | Explain how files are implemented | CO5 | K2 | 113 |
| 11 | Explain various file allocation methods | CO5 | K2 | 117 |
| 12 | Explain access matrix | CO5 | K2 | 122 |

**APPENDIX 1**

**CONTENT BEYOND THE SYLLABUS**

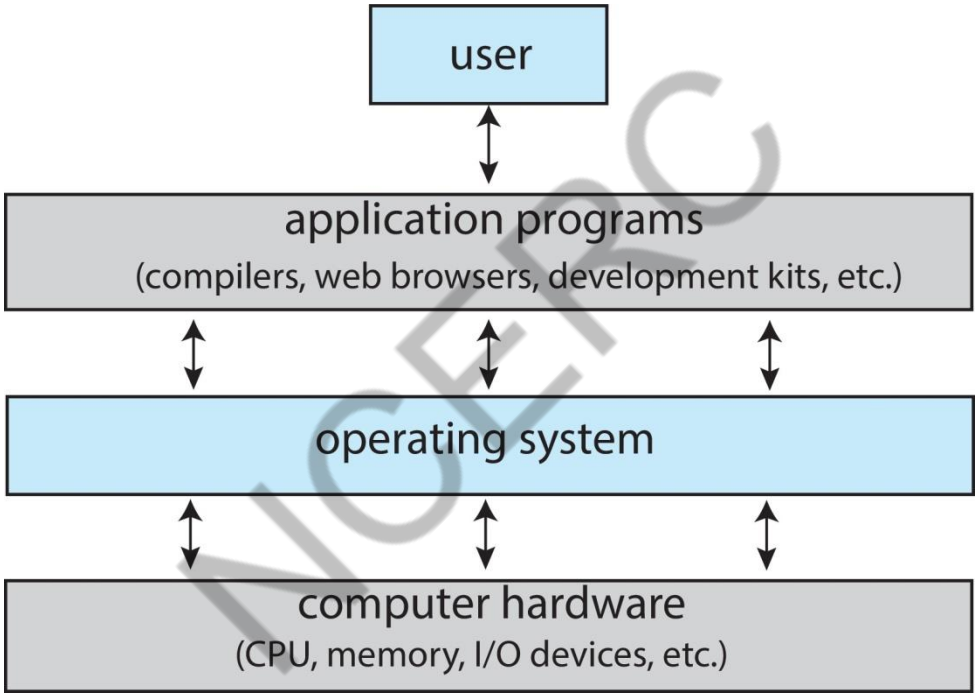
| S:NO; | TOPIC | PAGE NO: |
|---|---|---|
| 1 | Microsoft Windows | 124 |
| 2 | MacOS | 125 |
| 3 | Ubuntu | 126 |
| 4 | Linux Fedora, Linux mint | 127 |
| 5 | Elementary OS | 129 |
| 6 | Solaris | 130 |
| 7 | Solus | 131 |
| 8 | Chrome OS | 132 |
| 9 | CentOS | 133 |

# CST 206

# OPERATING SYSTEMS

# MODULE 1

An **operating system** is a program that manages a computer's hardware. It also provides a basis for application programs and acts as an intermediary between the computer user and the computer hardware. An amazing aspect of operating systems is how they vary in accomplishing these tasks. Mainframe operating systems are designed primarily to optimize utilization of hardware. Personal computer (PC) operating systems support complex games, business applications, and everything in between. Operating systems for mobile computers provide an environment in which a user can easily interface with the computer to execute programs. Thus, some operating systems are designed to be ***convenient,*** others to be ***efficient,*** and others to be some combination of the two




**Abstract View of Components of Computer**

## What Operating Systems Do

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.

### User View

The user's view of the computer varies according to the interface being used. Most computer users sit in front of a PC, consisting of a monitor, keyboard, mouse, and system unit. Such a system is designed for one user to monopolize its resources. The goal is to maximize the work (or play) that the user is performing. In this case, the operating system is designed mostly for **ease of use**, with some attention paid to performance and none paid to **resource utilization**—how various hardware and software resources are shared. Performance is, of course, important to the user; but such systems are optimized for the single-user experience rather than the requirements of multiple users In other cases, a user sits at a terminal connected to a **mainframe** or a **minicomputer**. Other users are accessing the same computer through other terminals. These users share resources and may exchange information. The operating system in such cases is designed to maximize resource utilization— to assure that all available CPU time, memory, and I/O are used efficiently and that no individual user takes more than her fair share. In still other cases, users sit at **workstations** connected to networks of other workstations and **servers**. These users have dedicated resources at their disposal, but they also share resources such as networking and servers, including file, compute, and print servers. Therefore, their operating system is designed to compromise between individual usability and resource utilization. The user interface for mobile computers generally features a **touch screen**, where the user interacts with the system by pressing and swiping fingers across the screen rather than using a physical keyboard and mouse. Some computers have little or no user view. For example, embedded computers in home devices and automobiles may have numeric keypads and may turn indicator lights on or off to show status, but they and their operating systems are designed primarily to run without user intervention

**System View**

From the computer's point of view, the operating system is the program most intimately involved with the hardware. In this context, we can view an operating system as a **resource allocator**. A computer system has many resources that may be required to solve a problem: CPU time, memory space, file-storage space, I/O devices, and so on. The operating system acts as the manager of these resources. Facing numerous and possibly conflicting requests for resources, the operating system must decide how to allocate them to specific programs and users so that it can operate the computer system efficiently and fairly. As we have seen, resource allocation is especially important where many

users access the same mainframe or minicomputer. A slightly different view of an operating system emphasizes the need to control the various I/O devices and user programs. An operating system is a control program. A **control program** manages the execution of user programs to prevent errors and improper use of the computer. It is especially concerned with the operation and control of I/O devices.

## Operating-System Operations

Modern operating systems are **interrupt driven**. If there are no processes to execute, no I/O devices to service, and no users to whom to respond, an operating system will sit quietly, waiting for something to happen. Events are almost always signaled by the occurrence of an interrupt or a trap. A **trap** (or an **exception**) is a software-generated interrupt caused either by an error (for example, division by zero or invalid memory access) or by a specific request from a user program that an operating-system service be performed. The interrupt-driven nature of an operating system defines that system's general structure. For each type of interrupt, separate segments of code in the operating system determine what action should be taken. An interrupt service routine is provided to deal with the interrupt Since the operating system and the users share the hardware and software resources of the computer system, we need to make sure that an error in a user program could cause problems only for the one program running. With sharing, many processes could be adversely affected by a bug in one program.

## Dual-Mode and Multimode Operation

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

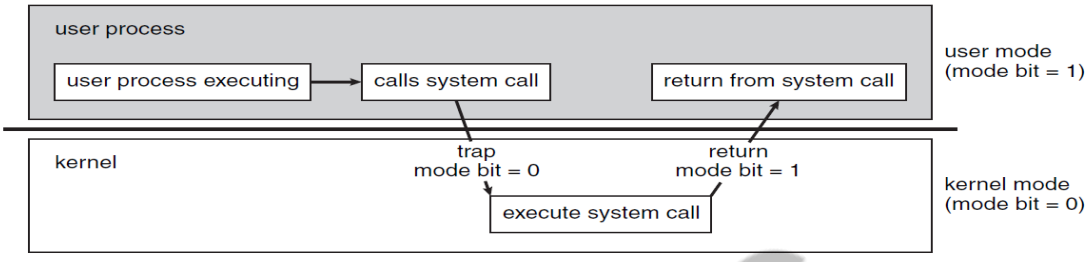



**Figure 1.10** Transition from user to kernel mode.

At the very least, we need two separate ***modes*** of operation: **user mode** and **kernel mode** (also called **supervisor mode**, **system mode**, or **privileged mode**). A bit, called the **mode bit**, is added to the hardware of the computer to indicate the current mode: kernel (0) or user (1). With the mode bit, we can distinguish between a task that is executed on behalf of the operating system and one that is executed on behalf of the user. When the computer system is executing on behalf of a user application, the system is in user mode. However, when a user application requests a service from the operating system (via a system call), the system must transition from user to kernel mode to fulfill the request. This is shown in Figure 1.10. As we shall see, this architectural enhancement is useful for many other aspects of system operation as well. At system boot time, the hardware starts in kernel mode. The operating system is then loaded and starts user applications in user mode. Whenever a trap or interrupt occurs, the hardware switches from user mode to kernel mode (that is, changes the state of the mode bit to 0). Thus, whenever the operating system gains control of the computer, it is in kernel mode. The system always switches to user mode (by setting the mode bit to 1) before passing control to a user program. The dual mode of operation provides us with the means for protecting the operating system from errant users—and errant users from one another. We accomplish this protection by designating some of the machine instructions that may cause harm as **privileged instructions**. The hardware allows privileged instructions to be executed only in kernel mode. If an attempt is made to execute a privileged instruction in user mode, the hardware does not execute the instruction but rather treats it as illegal and traps it to the operating system.

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 management, there are many additional privileged instructions 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. It needs that level of privilege so it can create and manage virtual machines, changing the CPU state to do so. Sometimes, too, different modes are used by various kernel components. We should note that, as an alternative to modes, the CPU designer may use other methods to differentiate operational privileges.

This trap can be executed by a generic trap instruction, although some systems (such as MIPS) have a specific syscall instruction to invoke a system call. When a system call is executed, it is typically treated by the hardware as a software interrupt. Control passes through the interrupt vector to a service routine in the operating system, and the mode bit is set to kernel mode. The system-call service routine is a part of the operating system. The kernel examines the interrupting instruction to determine what system call has occurred; a parameter indicates what type of service the user program is requesting. Additional information needed for the request may be passed in registers, on the stack, or in memory (with pointers to the memory locations passed in registers). The kernel verifies that the parameters are correct and legal, executes the request, and returns control to the instruction following the system call.



**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** is generally implemented by a fixed-rate clock and a counter. The operating system sets the counter. Every time the clock ticks, the counter is decremented. When the counter reaches 0, an interrupt occurs. For instance, a 10-bit counter with a 1-millisecond clock allows interrupts at intervals from 1 millisecond to 1,024 milliseconds, in steps of 1 millisecond. Before turning over control to the user, the operating system ensures that the timer is set to interrupt. If the timer interrupts, control transfers automatically to the operating system, which may treat the interrupt as a fatal error or

may give the program more time. Clearly, instructions that modify the content of the timer are privileged.

# FUNCTIONS OF OPRATING SYSTEM

## Process Management

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

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

## Memory Management

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

- Keeping track of which blocks of memory are currently in use, and by which processes.
- Determining which blocks of code and data to move into and out of memory, and when.
- Allocating and deallocating memory as needed. ( E.g. new, malloc )

## Storage Management

***File-System Management***

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

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

***Mass-Storage Management***

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

- Free disk space management
- Storage allocation
- Disk scheduling

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

***Caching***

- There are many cases in which a smaller higher-speed storage space serves as a cache, or temporary storage, for some of the most frequently needed portions of larger slower storage areas.
- The hierarchy of memory storage ranges from CPU registers to hard drives and external storage. ( See table below. )
- The OS is responsible for determining what information to store in what level of cache, and when to transfer data from one level to another.
- The proper choice of cache management can have a profound impact on system performance.
- Data read in from disk follows a migration path from the hard drive to main memory, then to the CPU cache, and finally to the registers before it can be used, while data being written follows the reverse path. Each step ( other than the registers ) will typically fetch more data than is immediately needed, and cache the excess in order to satisfy future requests faster. For writing, small amounts of data are frequently buffered until there is enough to fill an entire "block" on the next output device in the chain.
- The issues get more complicated when multiple processes ( or worse multiple computers ) access common data, as it is important to ensure that every access reaches the most up-to-date copy of the cached data ( amongst several copies in different cache levels. )

| 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 1.11 - Performance of various levels of storage*

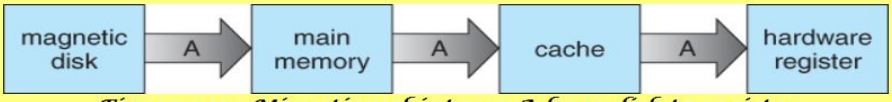



*Figure 1.12 - Migration of integer A from disk to register*

***I/O Systems***

The I/O subsystem consists of several components:

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

**OPERATING SYSTEM SERVICES**

OSes provide environments in which programs run, and services for the users of the system, including:

- **User Interfaces** - Means by which users can issue commands to the system. Depending on the system these may be a command-line interface ( e.g. sh, csh, ksh, tcsh, etc. ), a GUI interface ( e.g. Windows, X-Windows, KDE, Gnome, etc. ), or a batch command systems. The latter are generally older systems using punch cards of job-control

language, JCL, but may still be used today for specialty systems designed for a single purpose.

- **Program Execution** - The OS must be able to load a program into RAM, run the program, and terminate the program, either normally or abnormally.
- **I/O Operations** - The OS is responsible for transferring data to and from I/O devices, including keyboards, terminals, printers, and storage devices.
- **File-System Manipulation** - In addition to raw data storage, the OS is also responsible for maintaining directory and subdirectory structures, mapping file names to specific blocks of data storage, and providing tools for navigating and utilizing the file system.
- **Communications** - Inter-process communications, IPC, either between processes running on the same processor, or between processes running on separate processors or separate machines. May be implemented as either shared memory or message passing, ( or some systems may offer both. )
- **Error Detection** - Both hardware and software errors must be detected and handled appropriately, with a minimum of harmful repercussions. Some systems may include complex error avoidance or recovery systems, including backups, RAID drives, and other redundant systems. Debugging and diagnostic tools aid users and administrators in tracing down the cause of problems.



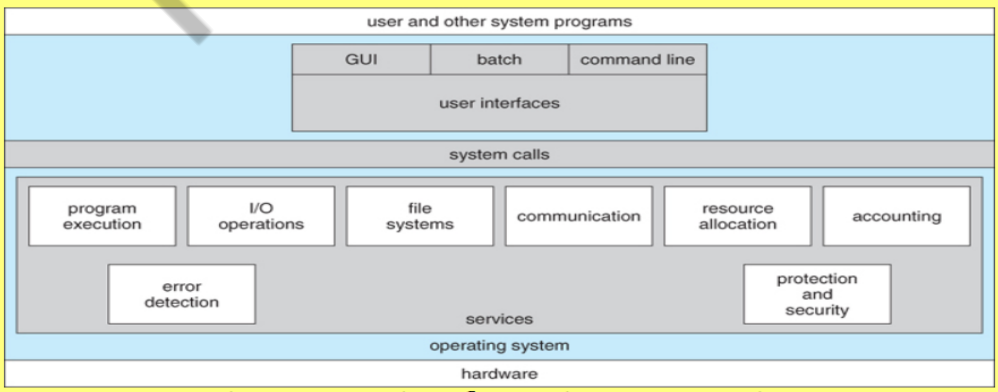



*Figure 2.1 - A view of operating system services*

Other systems aid in the efficient operation of the OS itself:

- **Resource Allocation** - E.g. CPU cycles, main memory, storage space, and peripheral devices. Some resources are managed with generic systems and others with very carefully designed and specially tuned systems, customized for a particular resource and operating environment.
- **Accounting** - Keeping track of system activity and resource usage, either for billing purposes or for statistical record keeping that can be used to optimize future performance.
- **Protection and Security** - Preventing harm to the system and to resources, either through wayward internal processes or malicious outsiders. Authentication, ownership, and restricted access are obvious parts of this system. Highly secure systems may log all process activity down to excruciating detail, and security regulation dictate the storage of those records on permanent non-erasable medium for extended times in secure ( off-site ) facilities.

## System Calls

- System calls provide a means for user or application programs to call upon the services of the operating system.
- Generally written in C or C++, although some are written in assembly for optimal performance.
- Figure 2.4 illustrates the sequence of system calls required to copy a file:

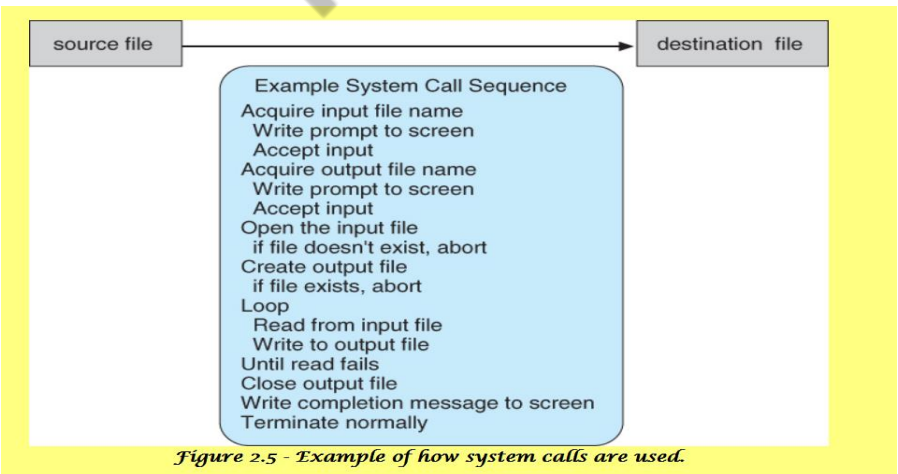



*Figure 2.5 - Example of how system calls are used.*

- You can use "strace" to see more examples of the large number of system calls invoked by a single simple command. Read the man page for strace, and try some simple examples. ( strace mkdir temp, strace cd temp, strace date > t.t, strace cp t.t t.2, etc. )

- Most programmers do not use the low-level system calls directly, but instead use an "Application Programming Interface", API. The following sidebar shows the read( ) call available in the API on UNIX based systems::

*EXAMPLE OF STANDARD API*

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

```
man read
```

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

```
#include <unistd.h>

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

return value — function name — parameters

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

- `int fd`—the file descriptor to be read
- `void *buf`—a buffer where the data will be read into
- `size_t count`—the maximum number of bytes to be read into the buffer

On a successful read, the number of bytes read is returned. A return value of 0 indicates end of file. If an error occurs, `read()` returns −1.



For most programming languages, the run-time support system (a set of functions built into libraries included with a compiler) provides a **systemcall 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. Typically, a number is associated with each system call, and the system-call interface maintains a table indexed according to these numbers. The system call interface

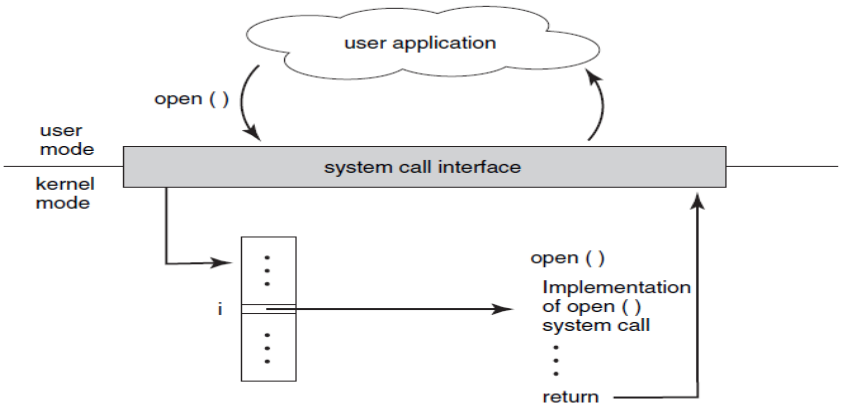



**Figure 2.6** The handling of a user application invoking the `open()` system call.

then invokes the intended system call in the operating-system kernel and returns the status of the system call and any return values. The caller need know nothing about how the system call is implemented or what it does during execution. Rather, the caller need only obey the API and understand what the operating system will do as a result of the execution of that system call. Thus, most of the details of the operating-system interface are hidden from the programmer by the API and are managed by the run-time support library. The relationship between an API, the system-call interface, and the operating system is shown in Figure 2.6, which illustrates how the operating system handles a user application invoking the open() system call. System calls occur in different ways, depending on the computer in use. Often, more information is required than simply the identity of the desired system call. The exact type and amount of information vary according to the particular operating system and call. For example, to get input, we may need to specify the file or device to use as the source, as well as the address and length of the memory buffer into which the input should be read.

Three general methods are used to pass parameters to the operating system. The simplest approach is to pass the parameters in registers. In some cases, however, there may be more parameters than registers. In these cases, the parameters are generally stored in a block, or table, in memory, and the address of the block is passed as a parameter in a register (Figure 2.7). This is the approach taken by Linux and Solaris. Parameters also can be placed, or **pushed**, onto the **stack** by the program and **popped** off the stack by the operating system. Some operating systems prefer the block or stack method because those approaches do not limit the number or length of parameters being passed.

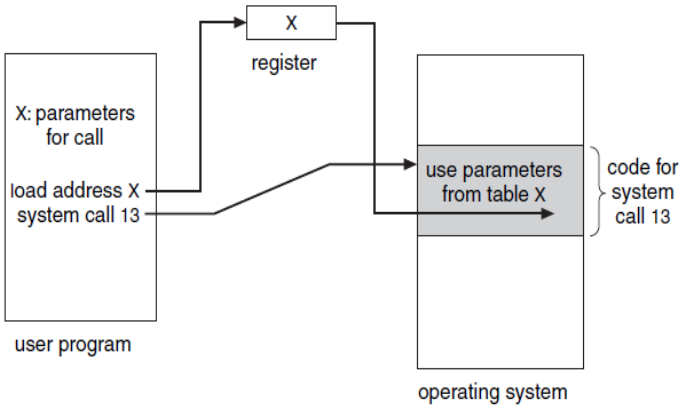



**Figure 2.7** Passing of parameters as a table.

## Types of System Calls

**Process control**

◦ end, abort

◦ load, execute

◦ create process, terminate process

◦ get process attributes, set process attributes

◦ wait for time

◦ wait event, signal event

◦ allocate and free memory

• **File management**

◦ create file, delete file

◦ open, close

◦ read, write, reposition

◦ get file attributes, set file attributes

• **Device management**

◦ request device, release device

◦ read, write, reposition

◦ get device attributes, set device attributes

◦ logically attach or detach devices

• **Information maintenance**

◦ get time or date, set time or date

◦ get system data, set system data

◦ get process, file, or device attributes

◦ set process, file, or device attributes

• **Communications**

◦ create, delete communication connection

◦ send, receive messages

◦ transfer status information

◦ attach or detach remote devices

*EXAMPLES OF WINDOWS AND UNIX SYSTEM CALLS*

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



## Operating-System Structure

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

### *2.7.1 Simple Structure*

When DOS was originally written its developers had no idea how big and important it would eventually become. It was written by a few programmers in a relatively short amount of time, without the benefit of modern software engineering techniques, and then gradually grew over time to exceed its original expectations. It does not break the system into subsystems, and has no

distinction between user and kernel modes, allowing all programs direct access to the underlying hardware. ( Note that user versus kernel mode was not supported by the 8088 chip set anyway, so that really wasn't an option back then. )

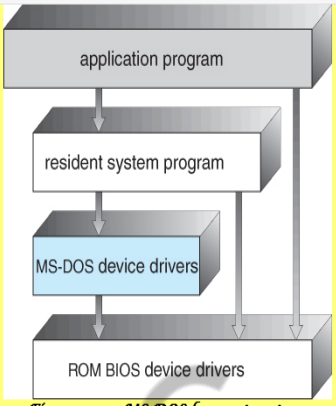



*Figure 2.11 - MS-DOS layer structure*

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

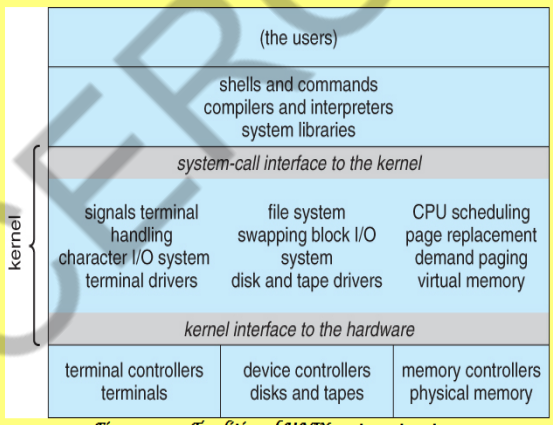



*Figure 2.12 - Traditional UNIX system structure*



***Layered Approach***

- Another approach is to break the OS into a number of smaller layers, each of which rests on the layer below it, and relies solely on the services provided by the next lower layer.
- This approach allows each layer to be developed and debugged independently, with the assumption that all lower layers have already been debugged and are trusted to deliver proper services.
- The problem is deciding what order in which to place the layers, as no layer can call upon the services of any higher layer, and so many chicken-and-egg situations may arise.

- Layered approaches can also be less efficient, as a request for service from a higher layer has to filter through all lower layers before it reaches the HW, possibly with significant processing at each step.

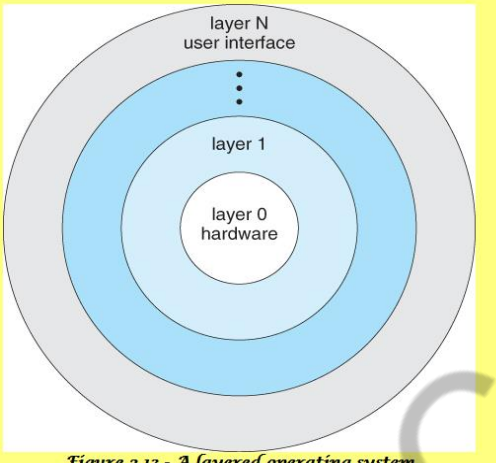



*Figure 2.13 - A layered operating system*

***Microkernels***

- The basic idea behind micro kernels is to remove all non-essential services from the kernel, and implement them as system applications instead, thereby making the kernel as small and efficient as possible.
- Most microkernels provide basic process and memory management, and message passing between other services, and not much more.
- Security and protection can be enhanced, as most services are performed in user mode, not kernel mode.
- System expansion can also be easier, because it only involves adding more system applications, not rebuilding a new kernel.
- Mach was the first and most widely known microkernel, and now forms a major component of Mac OSX.
- Windows NT was originally microkernel, but suffered from performance problems relative to Windows 95. NT 4.0 improved performance by moving more services into the kernel, and now XP is back to being more monolithic.
- Another microkernel example is QNX, a real-time OS for embedded systems.

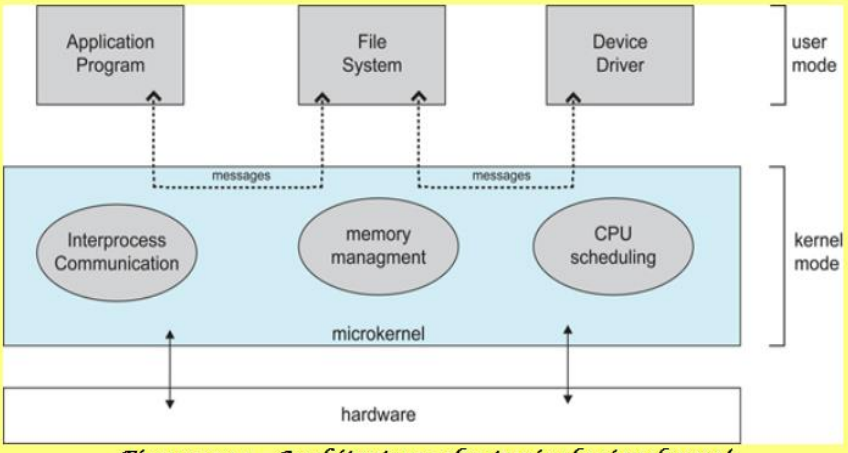



*Figure 2.14 - Architecture of a typical microkernel*

***Modules***

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

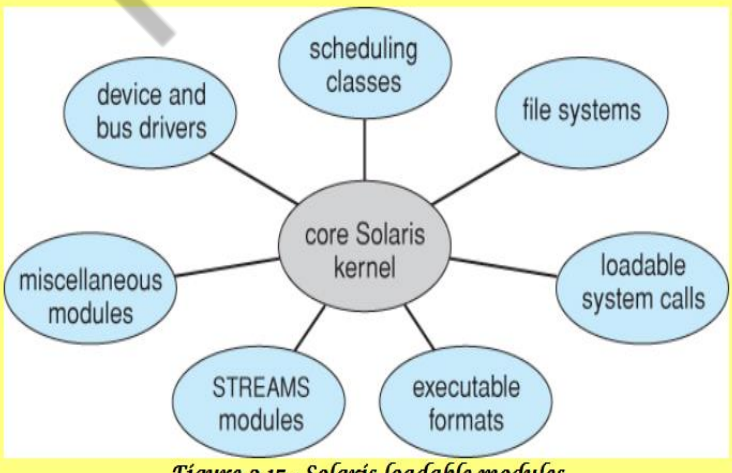



*Figure 2.15 - Solaris loadable modules*

## System Boot

The general approach when most computers boot up goes something like this:

- When the system powers up, an interrupt is generated which loads a memory address into the program counter, and the system begins executing instructions found at that address. This address points to the "bootstrap" program located in ROM chips ( or EPROM chips ) on the motherboard.
- The ROM bootstrap program first runs hardware checks, determining what physical resources are present and doing power-on self tests ( POST ) of all HW for which this is applicable. Some devices, such as controller cards may have their own on-board diagnostics, which are called by the ROM bootstrap program.
- The user generally has the option of pressing a special key during the POST process, which will launch the ROM BIOS configuration utility if pressed. This utility allows the user to specify and configure certain hardware parameters as where to look for an OS and whether or not to restrict access to the utility with a password.
  - Some hardware may also provide access to additional configuration setup programs, such as for a RAID disk controller or some special graphics or networking cards.
- Assuming the utility has not been invoked, the bootstrap program then looks for a non-volatile storage device containing an OS. Depending on configuration, it may look for a floppy drive, CD ROM drive, or primary or secondary hard drives, in the order specified by the HW configuration utility.
- Assuming it goes to a hard drive, it will find the first sector on the hard drive and load up the fdisk table, which contains information about how the physical hard drive is divided up into logical partitions, where each partition starts and ends, and which partition is the "active" partition used for booting the system.
- There is also a very small amount of system code in the portion of the first disk block not occupied by the fdisk table. This bootstrap code is the first step that is not built into the hardware, i.e. the first part which might be in any way OS-specific. Generally this code knows just enough to access the hard drive, and to load and execute a ( slightly ) larger boot program.

- For a single-boot system, the boot program loaded off of the hard disk will then proceed to locate the kernel on the hard drive, load the kernel into memory, and then transfer control over to the kernel. There may be some opportunity to specify a particular kernel to be loaded at this stage, which may be useful if a new kernel has just been generated and doesn't work, or if the system has multiple kernels available with different configurations for different purposes. ( Some systems may boot different configurations automatically, depending on what hardware has been found in earlier steps. )
- For dual-boot or multiple-boot systems, the boot program will give the user an opportunity to specify a particular OS to load, with a default choice if the user does not pick a particular OS within a given time frame. The boot program then finds the boot loader for the chosen single-boot OS, and runs that program as described in the previous bullet point.
- Once the kernel is running, it may give the user the opportunity to enter into single-user mode, also known as maintenance mode. This mode launches very few if any system services, and does not enable any logins other than the primary log in on the console. This mode is used primarily for system maintenance and diagnostics.
- When the system enters full multi-user multi-tasking mode, it examines configuration files to determine which system services are to be started, and launches each of them in turn. It then spawns login programs ( gettys ) on each of the login devices which have been configured to enable user logins.
  - ( The getty program initializes terminal I/O, issues the login prompt, accepts login names and passwords, and authenticates the user. If the user's password is authenticated, then the getty looks in system files to determine what shell is assigned to the user, and then "execs" ( becomes ) the user's shell. The shell program will look in system and user configuration files to initialize itself, and then issue prompts for user commands. Whenever the shell dies, either through logout or other means, then the system will issue a new getty for that terminal device. )

# CST 206

# OPERATING SYSTEMS

# MODULE II

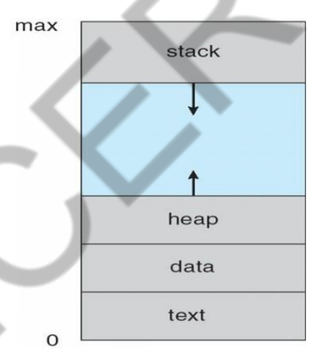
# MODULE II

## PROCESS CONCEPT

- Process is a program in execution. A process is the unit of work in a modern time-sharing system.
- An operating system executes a variety of programs:
  - Batch system – jobs
  - Time-shared systems – user programs or tasks

### The Process

- A process is more than the program code, it includes
  - The program code, also called **text section**
  - Current activity including program counter, processor registers
  - Stack containing temporary data
    Function parameters, return addresses, local variables
  - Data section containing global variables
  - Heap containing memory dynamically allocated during run time
- program is a *passive* entity(executable files), whereas a process is an *active* entity
- A program becomes a process when an executable file is loaded into memory.




Process in memory

### Process State

- As a process executes, it changes state.
- The state of a process is defined in part by the current activity of that process.
- Each 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
- Only one process can be *running* on any processor at any instant. Many processes may be *ready* and *waiting.*

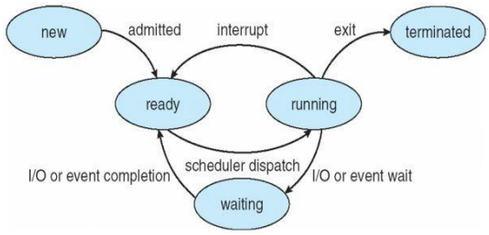
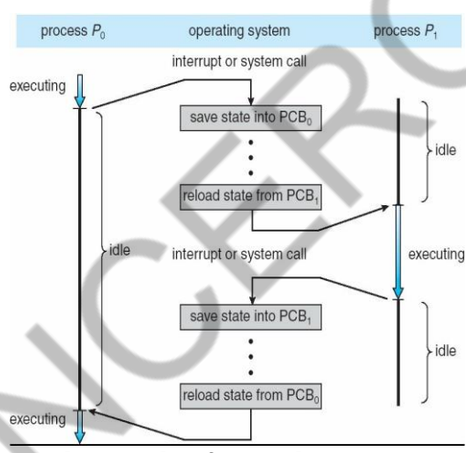



Diagram of process state

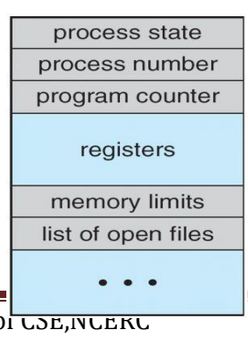
## Process Control Block

- Each process is represented on the OS by a Process Control Block(PCB) also called a *task control block.*.
- It includes
- Process state – The state may be new, ready running, waiting, halted etc
- Program counter – The counter indicates the address of the next instruction to be executed for this process
- CPU registers – The registers vary in number and type, depending on the computer architecture. They include accumulators, index registers, stack pointers, and general-purpose registers, plus any condition-code information. Along with the program counter, this state information must be saved when an interrupt occurs, to allow the




process to be continued correctly afterward

- CPU scheduling information- includes a process priority, pointers to scheduling queues, and any other scheduling parameters.
- Memory-management information -include such information as the value of the base and limit registers, the page tables, or the segment tables, depending on the memory system used by the operating system.
- Accounting 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 – I/O devices allocated to process, list of open files



Process Control Block

## Threads

The process model introduced earlier assumed that a process was an executing program with a single thread of control. Virtually all modern operating systems, however, provide features enabling a process to contain multiple threads of control. Here, we introduce many concepts associated with multithreaded computer systems, including a discussion of the APIs for the Pthreads, Windows, and Java thread libraries.

Overview

A thread is a basic unit of CPU utilization; it comprises a thread ID, a program counter, a register set, and a stack. It shares with other threads belonging to the same process its code section, data section, and other operating-system resources, such as open files and signals. A traditional (or *heavyweight*) process has a single thread of control. If a process has multiple threads of control, it can perform more than one task at a time

Figure 4.1 illustrate the difference between a traditional **single-threaded** process and a **multithreaded** process.

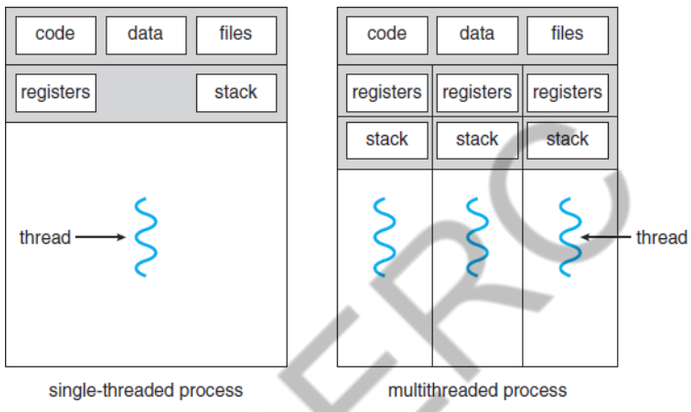



**Figure 4.1** Single-threaded and multithreaded processes.

- ➢ Process is a program that performs single thread of execution.
- ➢ Single thread of control allows the process to perform only one task at one time.
- ➢ Modern Operating Systems have extended the process concept to allow a process to have multiple threads of execution and thus to perform more than one task at a time.

## Benefits

The benefits of multithreaded programming can be broken down into four major categories:

1. **Responsiveness**. Multithreading an interactive application may allow a program to continue running even if part of it is blocked or is performing a lengthy operation, thereby increasing responsiveness to the user. This quality is especially useful in designing user interfaces. For instance, consider what happens when a user clicks a button that results in the performance of a time-consuming operation. A single-threaded application would be unresponsive to the user until the operation had completed. In contrast, if the time- consuming operation is performed in a separate thread, the application remains responsive to the user.
2. **Resource sharing**. Processes can only share resources through techniques such as shared memory and message passing. Such techniques must be explicitly arranged by the programmer. However, threads share the memory and the resources of the process to which they belong by default. The benefit of sharing code and data is that it allows an application to have several different threads of activity within the same address space.

3. **Economy**. Allocating memory and resources for process creation is costly. Because threads share the resources of the process to which they belong, it is more economical to create and context-switch threads. Empirically gauging the difference in overhead can be difficult, but in general it is significantly more time consuming to create and manage processes than threads. In Solaris, for example, creating a process is about thirty times slower than is creating a thread, and context switching is about five times slower

4. **Scalability.** The benefits of multithreading can be even greater in a multiprocessor architecture, where threads may be running in parallel on different processing cores. A single-threaded process can run on only one processor, regardless how many are available.

## Multithreading Models

So far has treated threads in a generic sense. However, support for threads may be provided either at the user level, for **user threads**, or by the kernel, for **kernel threads**. User threads are supported above the kernel and are managed without kernel support, whereas kernel threads are supported and managed directly by the operating system. Virtually all contemporary operating systems—including Windows, Linux, Mac OS X, and Solaris— support kernel threads. Ultimately, a relationship must exist between user threads and kernel threads. In this section, we look at three common ways of establishing such a relationship: the many-to-one model, the one-to-one model, and the many-to many model.



## Many-to-One Model

The many-to-one model **(Figure 4.5)** maps many user-level threads to one kernel thread. Thread management is done by the thread library in user space, so it is efficient. However, the entire process will block if a thread makes a blocking system call. Also, because only one thread can access the kernel at a time, multiple threads are unable to run in parallel on multicore systems.

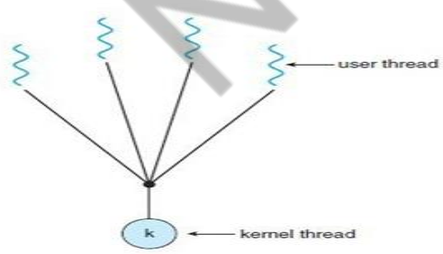



Figure 4.5 Many-to-one model.

**Green threads**—a thread library available for Solaris systems and adopted in early versions of Java—used the many-to-one model. However, very few systems continue to use the model because of its inability to take advantage of multiple processing cores.

## One-to-One Model

The one-to-one model (Figure 4.6) maps each user thread to a kernel thread. It provides more concurrency than the many-to-one model by allowing another thread to run when a thread makes a blocking system call. It also allows multiple threads to run in parallel on multiprocessors.

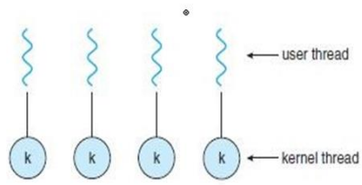



Figure 4.6 One-to-one model.

The only drawback to this model is that creating a user thread requires creating the corresponding kernel thread. Because the overhead of creating kernel threads can burden the performance of an application, most implementations of this model restrict the number of threads supported by the system. Linux, along with the family of Windows operating systems, implement the one-to-one model.

## Many-to-Many Model

The many-to-many model (Figure 4.7) multiplexes many user-level threads to a smaller or equal number of kernel threads. The number of kernel threads may be specific to either a particular application or a particular machine (an application may be allocated more kernel threads on a multiprocessor than on a single processor). Let's consider the effect of this design on concurrency Whereas the many to- one model allows the developer to create as many user threads as she wishes,

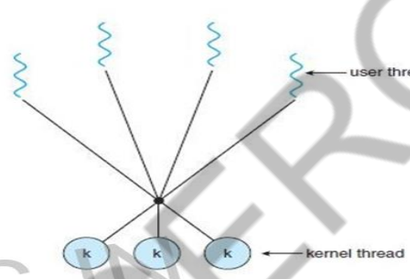



Figure 4.7 Many-to-many model.

it does not result in true concurrency, because the kernel can schedule only one thread at a time. The one-to-one model allows greater concurrency, but the developer has to be careful not to create too many threads within an application (and in some instances may be limited in the number of threads she can create). The many-to many model suffers from neither of these shortcomings: developers can create as many user threads as necessary, and the corresponding kernel threads can run in parallel on a multiprocessor. Also, when a thread performs a blocking system call, the kernel can schedule another thread for execution.

One variation on the many-to-many model still multiplexes many user level threads to a smaller or equal number of kernel threads but also allows a user-level thread to be bound to a kernel thread. This variation is sometimes referred to as the **two-level model**

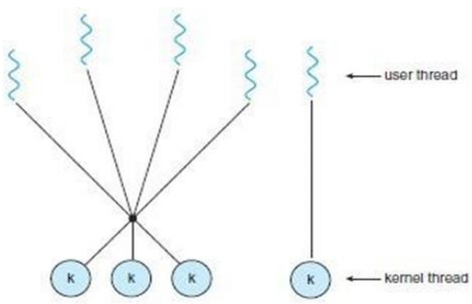



Figure 4.8 Two-level model.

## PROCESS SCHEDULING

➢ The objective of time sharing is to switch the CPU among processes so frequently that users can interact with each program while it is running.

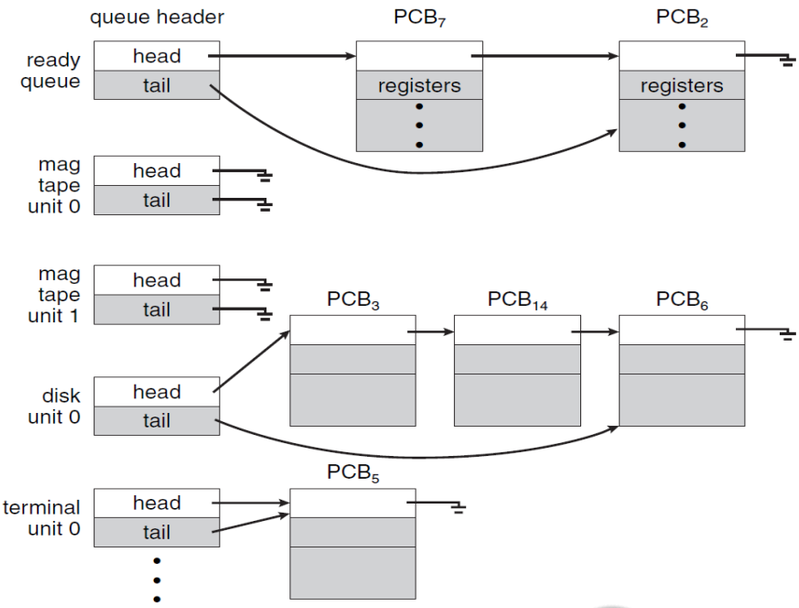


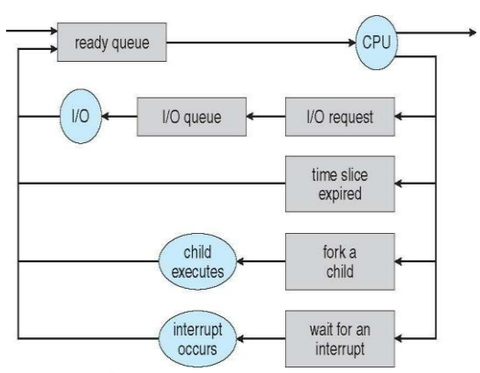

**Figure 3.5** The ready queue and various I/O device queues.

To meet these objectives, the **process scheduler** selects an available process (possibly from a set of several available processes) for program execution on the CPU. For a single-processor system, there will never be more than one running process

➢ The process scheduler selects an available process for program execution on the CPU.

➢ Maintains scheduling queues of processes

- Job queue – set of all processes in the system
- Ready queue – set of all processes residing in main memory, ready and waiting to execute
- Device queues – set of processes waiting for an I/O device




- Processes migrate among the various queues

Queuing diagram representation of process scheduling

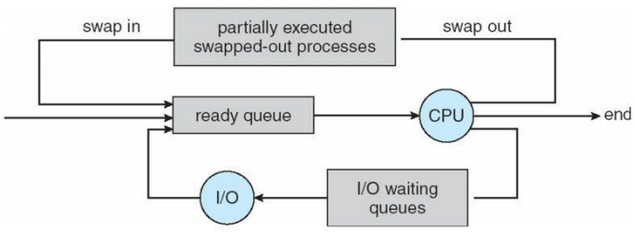
➢ A new process is initially put in the ready queue. It waits there until it is selected for execution, or is dispatched. Once the process is allocated the CPU and is executing, one of several events could occur:

➢ The process could issue an I/0 request and then be placed in an I/0 queue.

- The process could create a new subprocess and wait for the subprocess's termination.
- The process could be removed forcibly from the CPU, as a result of an interrupt, and be put back in the ready queue

## Schedulers

- A process migrates among the various scheduling queues throughout its lifetime.
- The operating system must select, for scheduling purposes, processes from these queues in some fashion.
- The selection process is carried out by the appropriate scheduler
- The long-term scheduler, or job scheduler, selects processes from pool and loads them into memory for execution. Long-term scheduler is invoked infrequently (seconds, minutes).
- The short-term scheduler, or CPU scheduler, selects from among the processes that are ready to execute and allocates the CPU to one of them. Short-term scheduler is invoked frequently (milliseconds) ⇒(must be fast)
- Processes can be described as either:
- I/O-bound process – spends more time doing I/O than computations, many short CPU bursts
- CPU-bound process – spends more time doing computations; few very long CPU bursts
- Long-term scheduler strives for good *process mix* of I/O-bound and CPU-bound processes.
- **Medium-term scheduler** can be added if degree of multiple programming needs to decrease
- **Swapping** -Remove process from memory, store on disk, bring back in from disk to continue execution




Addition of medium-term scheduling to the queuing diagram

## Context Switch

- When CPU switches to another process, the system must **save the state** of the old process and load the **saved state** for the new process via a **context switch**
- **Context** of a process represented in the PCB

- ➢ Context-switch time is overhead; the system does no useful work while switching
- ➢ The more complex the OS and the PCB ➔the longer the context switch
- ➢ Time dependent on hardware support
- ➢ Some hardware provides multiple sets of registers per CPU ➔multiple contexts loaded at once

## Operations on Processes

The processes in most systems can execute concurrently, and they may be created and deleted dynamically. Thus, these systems must provide a mechanism for process creation and termination.

### Process Creation

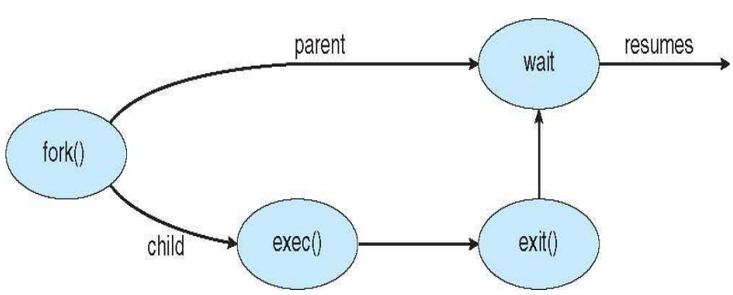
- ➢ A process may create several new processes, via a create-process system call,during the course of execution.
- ➢ The creating process is called a parent process, and the new processes are called the children of that process. Each of these new processes may in turn create other processes, forming a tree of processes.
- ➢ Generally, process identified and managed via a process identifier (pid)
- ➢ Resource sharing options
  - ➢ Parent and children share all resources
  - ➢ Children share subset of parent's resources
  - ➢ Parent and child share no resources

Execution

- o Parent and children execute concurrently
- o Parent waits until children terminate

Possibilities in terms of the address space of the new process

- ➢ The child process is a duplicate of the parent process
- ➢ The child process has a new program loaded into it.
- ➢ **fork()** system call creates new process
- ➢ **exec()** system call used after a **fork()** to replace the process' memory space with a new program




Process creation using the fork() system call.

**C program forking a separate process**

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

int main()
{
pid_t pid;

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

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

    return 0;
}
```

## Process Termination

- Process executes last statement and then asks the operating system to delete it using the **exit()** system call
- Returns status data from child to parent (via **wait()**)
- Process' resources are deal located by operating system
- A process can cause the termination of another process via an appropriate system call. Such a system call can be invoked only by the parent of the process that is to be terminated
- A parent needs to know the identities of its children. Thus, when one process creates a new process, the identity of the newly created process is passed to the parent.
- A parent may terminate the execution of one of its children for a variety of reasons, such as these:
- Child has exceeded allocated resources
- Task assigned to child is no longer required

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. (To determine whether this has occurred, the parent must have a mechanism to inspect the state of its children.)
- 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

➢ The parent is exiting and the operating systems does not allow a child to continue if its parent terminates

➢ Some operating systems do not allow child to exist if its parent has terminated. If a process terminates, then all its children must also be terminated. This phenomenon, referred to as **cascading termination.** The termination is initiated by the operating system

➢ The parent process may wait for termination of a child process by using the **wait()**system call**.** The call returns status information and the pid of the terminated process



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. All processes transition to this state when they terminate, but generally they exist as zombies only briefly. Once the parent calls wait(), the process identifier of the zombie process and its entry in the process table are released.

## INTERPROCESS COMMUNICATION

➢ Inter process communication (IPC) refers to the coordination of activities among cooperating processes.

:

There are several reasons for providing an environment that allows 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.

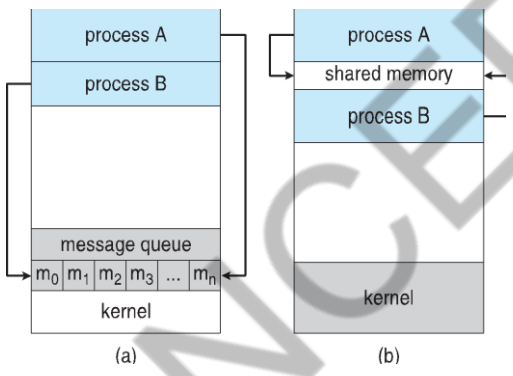
• **Computation speedup**. If we want a particular task to run faster, we must break it into subtasks, each of which will be executing in parallel with the others. Notice that such a speedup can be achieved only if the computer has multiple processing cores.

• **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.

➢ Two models of IPC

- **Shared memory-**
- **Message passing**




Communications models. (a) Message passing. (b) Shared memory



## Shared-Memory Systems

➢ Interprocess communication using shared memory requires communicating processes to establish a region of shared memory.

➢ A shared-memory region resides in the address space of the process creating the shared memory segment. Other processes that wish to communicate using this shared memory segment must attach it to their address space.

➢ They can then exchange information by reading and writing data in the shared areas.

➢ The communication is under the control of the users processes not the operating system.

➢ Example for cooperating processes.:-Producer-consumer problem.

- A producer process produces information that is consumed by a consumer process.
- One solution to the producer-consumer problem uses shared memory.
- A buffer which reside in a region of memory that is shared by the producer and consumer processes is used
- 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.
  - **unbounded-buffer** places no practical limit on the size of the buffer
  - **bounded-buffer** assumes that there is a fixed buffer size

➢ **Bounded-Buffer – Shared-Memory Solution**

➢ Shared data

```
#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.

➢ Producer

```
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;
}
```

➢ Consumer

```
item next_consumed;

while(true){
        while (in == out)
                ;/*donothing*/
        next_consumed =
        buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        /* consume the item in next consumed */
}
```

**Message-Passing Systems**

➢ 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) and receive (message).

➢ 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.

➢ 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.
- Symmetry in addressing- both the sender process and the receiver process must name the other to communicate.
- Asymmetry in addressing- Here, only the sender names the recipient; the recipient is not required to name the sender.
  - send (P, message) -Send a message to process P.
  - receive (id, message) -Receive a message from any process
- With indirect communication, the messages are sent to and received from mailboxes, or ports.
- Two processes can communicate only if the processes have a shared mailbox.
  - send (A, message) -Send a message to mailbox A.
  - receive (A, message)-Receive a message from mailbox A.
- The operating system must provide a mechanism that allows a process to do the following:
  - Create a new mailbox.
  - Send and receive messages through the mailbox.
  - Delete a mailbox.

**Synchronization**

- Message passing may be either blocking or nonblockingalso 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. 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.



**Preemptive Scheduling**

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

**1.** When a process switches from the running state to the waiting state (for example, as the result of an I/O request or an invocation of wait() for the termination of a child process)

**2**. When a process switches from the running state to the ready state (for example, when an interrupt occurs)

**3.** When a process switches from the waiting state to the ready state (for example, at completion of I/O)

**4.** When a process terminates

For situations 1 and 4, there is no choice in terms of scheduling. A new process (if one exists in the ready queue) must be selected for execution. There is a choice, however, for situations 2 and 3. When scheduling takes place only under circumstances 1 and 4, we say that the scheduling scheme is

**nonpreemptive** or **cooperative**. Otherwise, it is **preemptive**. Under nonpreemptive scheduling, once the CPU has been allocated to a process, the process keeps the CPU until it releases the CPU either by

terminating or by switching to the waiting state. This scheduling method was used by Microsoft Windows 3.x. Windows 95 introduced preemptive scheduling, and all subsequent versions of Windows operating systems have used preemptive scheduling

### Dispatcher

Another component involved in the CPU-scheduling function is the **dispatcher**. The dispatcher is the module that gives control of the CPUto the process selected by the short-term scheduler. This function involves the following:

• Switching context

• Switching to user mode

• Jumping to the proper location in the user program to restart that program

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

## Scheduling Criteria

Many criteria have been suggested for comparing CPU-scheduling algorithms. Which characteristics are used for comparison can make a substantial difference in which algorithm is judged to be best. The criteria include the following:



- **CPU utilization**. We want to keep the CPU as busy as possible. Conceptually, CPU utilization can range from 0 to 100 percent. In a real system, it should range from 40 percent (for a lightly loaded system) to 90 percent (for a heavily loaded system).

• **Throughput**. If the CPU is busy executing processes, then work is being done. One measure of work is the number of processes that are completed per time unit, called **throughput**. For long processes, this rate may be one process per hour; for short transactions, it may be ten processes per second.

• **Turnaround time**. From the point of view of a particular process, the important criterion is how long it takes to execute that process. The interval from the time of submission of a process to the time of completion is the turnaround time. Turnaround time is the sum of the periods spent waiting to get into memory, waiting in the ready queue, executing on the CPU, and doing I/O.

• **Waiting time**. The CPU-scheduling algorithm does not affect the amount of time during which a process executes or does I/O. It affects only the amount of time that a process spends waiting in the ready queue.Waiting time is the sum of the periods spent waiting in the ready queue.

• **Response time**. In an interactive system, turnaround time may not be the best criterion. Often, a process can produce some output fairly early and can continue computing new results while previous results are being output to the user. Thus, another measure is the time from the submission of a request until the first response is produced. This measure, called response time, is the time it takes to start responding, not the time it takes to output the response. The turnaround time is generally limited by the speed of the output device.

## Scheduling Algorithms

CPU scheduling deals with the problem ofdeciding which of the processes in the ready queue is to be allocated the CPU.There are many different CPU-scheduling algorithms

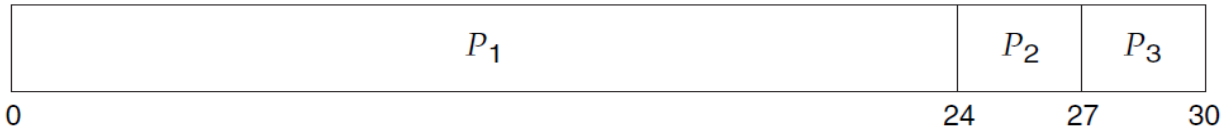
**First-Come, First-Served Scheduling**

By far the simplest CPU-scheduling algorithm is the **first-come, first-served (FCFS)** scheduling algorithm. With this scheme, the process that requests the CPU first is allocated the CPU first. The implementation of the FCFS policy is easily managed with a FIFO queue. When a process enters the ready queue, its PCB is linked onto the tail of the queue. When the CPU is free, it is allocated to the process at the head of the queue. The running process is then removed from the queue. The code for FCFS scheduling is simple to write and understand. On the negative side, the average waiting time under the FCFS policy is

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

| Process | Burst Time |
|---|---|
| $P_1$ | 24 |
| $P_2$ | 3 |
| $P_3$ | 3 |

If the processes arrive in the order $P_1$, $P_2$, $P_3$, and are served in FCFS order, we get the result shown in the following **Gantt chart**, which is a bar chart that illustrates a particular schedule, including the start and finish times of each of the participating processes:




The waiting time is 0 milliseconds for process $P_1$, 24 milliseconds for process $P_2$, and 27 milliseconds for process $P_3$. Thus, the average waiting time is $(0 + 24 + 27)/3 = 17$ milliseconds. If the processes arrive in the order $P_2$, $P_3$, $P_1$, however, the results will be as shown in the following Gantt chart:


The average waiting time is now $(6 + 0 + 3)/3 = 3$ milliseconds. This reduction is substantial. Thus, the average waiting time under an FCFS policy is generally not minimal and may vary substantially if the processes' CPU burst times vary greatly.

In addition, consider the performance of FCFS scheduling in a dynamic situation. Assume we have one CPU-bound process and many I/O-bound processes. As the processes flow around the system, the following scenario may result. The CPU-bound process will get and hold the CPU. During this time, all the other processes will finish their I/O and will move into the ready queue, waiting for the CPU. While the processes wait in the ready queue, the I/O devices are idle. Eventually, the CPU-bound process finishes its CPU burst and moves to an I/O device. All the I/O-bound processes, which have short CPU bursts, execute quickly and move back to the I/O queues. At this point, the CPU sits idle. The CPU-bound process will then move back to the ready queue and be allocated the CPU. Again, all the I/O processes end up waiting in the ready queue until the CPU-bound process is done. There is a **convoy effect** as all the other processes wait for the one big process to get off the CPU. This effect results in lower CPU and device utilization than might be possible if the shorter processes were allowed to go first.

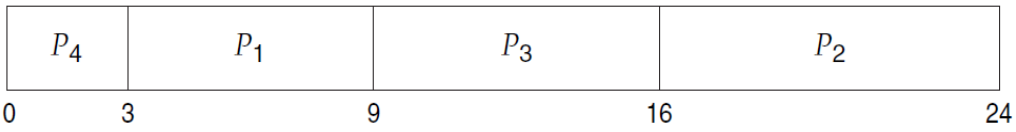
The FCFS scheduling algorithm is non-preemptive. Once the CPU has been allocated to a process, that process keeps the CPU until it releases the CPU, either by terminating or by requesting I/O. The FCFS algorithm is thus particularly troublesome for time-sharing systems, where it is important that each user get a share of the CPU at regular intervals. It would be disastrous to allow one process to keep the CPU for an extended period.

**Shortest-Job-First Scheduling**

This algorithm associates with each process the length of the process's next CPU burst. When the CPU is available, it is assigned to the process that has the smallest next CPU burst. If the next CPU bursts of two processes are the same, FCFS scheduling is used to break the tie. Note that a more appropriate term for this scheduling method would be the ***shortest-next- CPU-burst*** algorithm, because scheduling depends on the length of the next CPU burst of a process, rather than its total length.We use the term SJF because most people and textbooks use this term to refer to this type of scheduling. As an example of SJF scheduling, consider the following set of processes, with the length of the CPU burst given in milliseconds:

| Process | Burst Time |
|---|---|
| $P_1$ | 6 |
| $P_2$ | 8 |
| $P_3$ | 7 |
| $P_4$ | 3 |

Using SJF scheduling, we would schedule these processes according to the following Gantt chart:




The waiting time is 3 milliseconds for process *P*1, 16 milliseconds for process *P*2, 9 milliseconds for process *P*3, and 0 milliseconds for process *P*4. Thus, the average waiting time is (3 + 16 + 9 + 0)/4 = 7 milliseconds. By comparison, if we were using the FCFS scheduling scheme, the average waiting time would be 10.25 milliseconds.

The SJF scheduling algorithm is provably optimal, in that it gives the minimum average waiting time for a given set of processes. Moving a short process before long one decrease the waiting time of the short

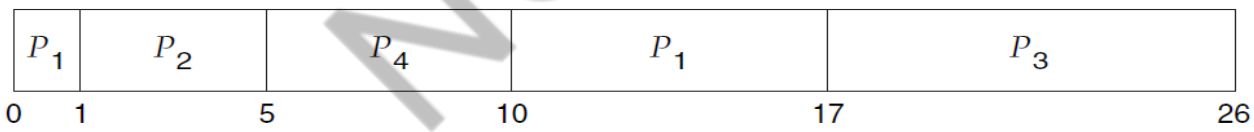
process more than it increases the waiting time of the long process. And thus the average waiting time decreases. The real difficulty with the SJF algorithm is knowing the length of the next CPU request. For long-term (job) scheduling in a batch system, we can use the process time limit that a user specifies when he submits the job. In this situation, users are motivated to estimate the process time limit accurately, since a lower value may mean faster response but too low a value will cause a time-limit-exceeded error and require resubmission. SJF scheduling is used frequently in long-term scheduling.

Preemptive SJF scheduling is sometimes called **shortest-remaining-time-first** scheduling

As an example, consider the following four processes, with the length of the CPU burst given in milliseconds:

| Process | Arrival Time | Burst Time |
|---|---|---|
| $P_1$ | 0 | 8 |
| $P_2$ | 1 | 4 |
| $P_3$ | 2 | 9 |
| $P_4$ | 3 | 5 |

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




Process $P_1$ is started at time 0, since it is the only process in the queue. Process $P_2$ arrives at time 1. The remaining time for process $P_1$ (7 milliseconds) is larger than the time required by process $P_2$ (4 milliseconds), so process $P_1$ is preempted, and process $P_2$ is scheduled. The average waiting time for this example is $[(10 - 1) + (1 - 1) + (17 - 2) + (5 - 3)]/4 = 26/4 = 6.5$ milliseconds. Nonpreemptive SJF scheduling would result in an average waiting time of 7.75 milliseconds.

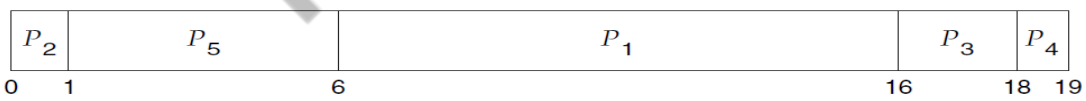
**Priority Scheduling**

The SJF algorithm is a special case of the general **priority-scheduling** algorithm. Apriority is associated with each process, and the CPUis allocated to the process with the highest priority. Equal-priority processes are scheduled in FCFS order. An SJF algorithm is simply a priority algorithm where the

priority ($p$) is the inverse of the (predicted) next CPU burst. The larger the CPU burst, the lower the priority, and vice versa. Note that we discuss scheduling in terms of ***high*** priority and ***low*** priority.
Priorities are generally indicated by some fixed range of numbers, such as 0 to 7 or 0 to 4,095. However, there is no general agreement on whether 0 is the highest or lowest priority. Some systems use low numbers to represent low priority; others use low numbers for high priority. This difference can lead to confusion. In this text, we assume that low numbers represent high priority. As an example, consider the following set of processes, assumed to have arrived at time 0 in the order $P1, P2, \cdots, P5$, with the length of the CPU burst given in milliseconds:

| Process | Burst Time | Priority |
|---|---|---|
| $P_1$ | 10 | 3 |
| $P_2$ | 1 | 1 |
| $P_3$ | 2 | 4 |
| $P_4$ | 1 | 5 |
| $P_5$ | 5 | 2 |

Using priority scheduling, we would schedule these processes according to the following Gantt chart:




The average waiting time is 8.2 milliseconds.

Priorities can be defined either internally or externally. Internally defined priorities use some measurable quantity or quantities to compute the priority of a process. For example, time limits, memory requirements, the number of open files, and the ratio of average I/O burst to average CPU burst have been used in computing priorities. Priority scheduling can be either preemptive or nonpreemptive. When a process arrives at the ready queue, its priority is compared with the priority of the currently running process. A preemptive priority scheduling algorithm will preempt the CPU if the priority of the newly

arrived process is higher than the priority of the currently running process. A nonpreemptive priority scheduling algorithm will simply put the new process at the head of the ready queue.

A major problem with priority scheduling algorithms is **indefinite blocking**, or **starvation**. A process that is ready to run but waiting for the CPU can be considered blocked. A priority scheduling algorithm

can leave some lowpriority processes waiting indefinitely. In a heavily loaded computer system, a steady stream of higher-priority processes can prevent a low-priority process from ever getting the CPU. Generally, one of two things will happen. Either the process will eventually be run or the computer system will eventually crash and lose all unfinished low-priority processes

A solution to the problem of indefinite blockage of low-priority processes is **aging**. Aging involves gradually increasing the priority of processes that wait in the system for a long time. For example, if priorities range from 127 (low) to 0 (high), we could increase the priority of a waiting process by 1 every 15 minutes. Eventually, even a process with an initial priority of 127 would have the highest priority in the system and would be executed.

**Round-Robin Scheduling**

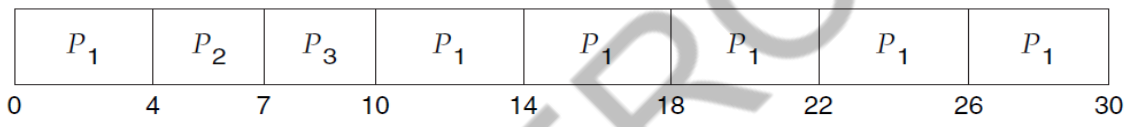
The **round-robin (RR)** scheduling algorithm is designed especially for timesharing systems. It is similar to FCFS scheduling, but preemption is added to enable the system to switch between processes. A small unit of time, called a **time quantum** or **time slice**, is defined. A time quantum is generally from 10 to 100 milliseconds in length. The ready queue is treated as a circular queue. The CPU scheduler goes around the ready queue, allocating the CPU to each process for a time interval of up to 1 time quantum.

To implement RR scheduling, we again treat the ready queue as a FIFO queue of processes. New processes are added to the tail of the ready queue. The CPU scheduler picks the first process from the ready queue, sets a timer to interrupt after 1 time quantum, and dispatches the process. One of two things will then happen. The process may have a CPU burst of less than 1 time quantum. In this case, the process itself will release the CPU voluntarily. The scheduler will then proceed to the next process in the ready queue. If the CPU burst of the currently running process is longer than 1 time quantum, the timer will go off and will cause an interrupt to the operating system. A context switch will be executed, and the process will be put at the tail of the ready queue. The CPU scheduler will then select the next process in the ready queue.

The average waiting time under the RR policy is often long. Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds:

| Process | Burst Time |
|---|---|
| $P_1$ | 24 |
| $P_2$ | 3 |
| $P_3$ | 3 |

If we use a time quantum of 4 milliseconds, then process $P_1$ gets the first 4 milliseconds. Since it requires another 20 milliseconds, it is preempted after the first time quantum, and the CPU is given to the next process in the queue, process $P_2$. Process $P_2$ does not need 4 milliseconds, so it quits before its time quantum expires. The CPU is then given to the next process, process $P_3$. Once each process has received 1 time quantum, the CPU is returned to process $P_1$ for an additional time quantum. The resulting RR schedule is as follows:




Let's calculate the average waiting time for this schedule. *P*1 waits for 6 milliseconds (10 - 4), *P*2 waits for 4 milliseconds, and *P*3 waits for 7 milliseconds. Thus, the average waiting time is 17/3 = 5.66 milliseconds. In the RR scheduling algorithm, no process is allocated the CPU for more than 1 time quantum in a row (unless it is the only runnable process). If a process's CPU burst exceeds 1 time quantum, that process is preempted and is put back in the ready queue. The RR scheduling algorithm is thus preemptive. If there are *n* processes in the ready queue and the time quantum is *q*, then each process gets 1/*n* of the CPU time in chunks of at most *q* time units. Each process must wait no longer than $(n - 1) \times q$ time units until its next time quantum. For example, with five processes and a time quantum of 20 milliseconds, each process will get up to 20 milliseconds every 100 milliseconds.

# CST 206

# OPERATING SYSTEMS

# MODULE III

## PROCESS SYNCHRONISATION

### CRITICAL SECTION

- **Each process has a segment of code, called a critical section in which the process may be changing common variables, updating a table, writing a file, and so on.**
- The important feature of the system is that, when one process is executing in its critical section, no other process is to be allowed to execute in its critical section.
- That is, no two processes are executing in their critical sections at the same time.

### Critical Section Problem

- The critical-section problem is to design a protocol that the processes can use to cooperate. Each process must request permission to enter its critical section. The section of code implementing this request is the entry section. The critical section may be followed by an exit section. The remaining code is the remainder section.
- The general structure of a typical process *Pi* is shown below

```
do
    {
        entry section

    critical section

        exit section

    remainder section

    } while (TRUE);
```

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

1. **Mutual exclusion.**

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

2. **Progress.**

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

**3 Bounded waiting.**

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

We assume that each process is executing at a nonzero speed. However, we can make no assumption concerning the relative speed of the *n* processes. At a given point in time, many kernel-mode processes may be active in the operating system. As a result, the code implementing an operating system *(kernel code)* is subject to several possible race conditions. Consider as an example a kernel data structure that maintains a list of all open files in the system. This list must be modified when a new file is opened or closed (adding the file to the list or removing it from the list). If two processes were to open files simultaneously, the separate updates to this list could result in a race condition. Other kernel data

structures that are prone to possible race conditions include structures for maintaining memory allocation, for maintaining process lists, and for interrupt handling.

- Two general approaches are used to handle critical sections in operating systems:
  - **(1) pre-emptive kernels**
  - **(2) nonpreemptive kernels**

- A pre-emptive kernel allows a process to be pre-empted while it is running in kernel mode.
- A nonpreemptive kernel does not allow a process running in kernel mode to be pre-empted.A kernel-mode process will run until it exits kernel mode, blocks, or voluntarily yields control of the CPU.

**PETERSON'S SOLUTION**

- A classic software-based solution to the critical-section problem known as Peterson's solution.
- Modern computer architectures perform basic machine-language instructions, such as load and store.
- There are no guarantees that Peterson's solution will work correctly on such architectures.
- We present the solution because it provides a good algorithmic description of solving the critical-section problem
- It illustrates some of the complexities involved in designing software that addresses the requirements of mutual exclusion, progress, and bounded waiting.
- Peterson's solution is restricted to two processes that alternate execution between their critical sections and remainder sections.
- The processes are numbered *Po* and P1. For convenience, when presenting *Pi*, we use *Pj* to denote the other process; that is, j equals 1 - i. Peterson's solution requires the two processes to share two data items:

  int turn; boolean flag[2];

The variable turn indicates whose turn it is to enter its critical section. That is, if turn == i, then process *Pi* is allowed to execute in its critical section. The flag array is used to indicate if a process *is ready* to enter its critical section. For example, if flag [i] is true, this value indicates that *Pi* is ready to enter its critical section. With an explanation of these data structures complete. The algorithm explains below.

To enter the critical section, process *Pi* first sets flag [i] to be true and then sets turn to the value j, thereby asserting that if the other process wishes to enter the critical section, it can do so. If both processes try to enter at the same time, turn will be set to both i and j at roughly the same time. Only one of these assignments will last; the other will occur but will be overwritten immediately

**Do**

{

```
flag [i] = TRUE;
turn= j;

while (flag[i] && turn i);
```

critical section

```
flag [i] = FALSE;
```

remainder section

}

While (TRUE);

The eventual value of turn determines which of the two processes is allowed to enter its critical section first.

To prove that this solution is correct, it is required to show that:

1 Mutual exclusion is preserved.

2 The progress requirement is satisfied.

3 The bounded-waiting requirement is met.

To prove property 1, we note that each *P;* enters its critical section only if either flag [j] == false or turn == i. Also note that, if both processes can be executing in their critical sections at the same time, then flag [0] == flag [1] ==true. These two observations imply that *Po* and P1 could not have successfully executed their while statements at about the same time, since the value of turn can be either 0 or 1 but cannot be both. Hence, one of the processes -say, *Pi* - must have successfully executed the while statement , whereas *P;* had *to* execute at least one additional statement ("turn== j"). However, at that time, flag [j] == true and turn == j, and this condition will persist as long as *Pi* is in its critical section; as a result, mutual exclusion is preserved. To prove properties 2 and 3, we note that a process *P;* can be prevented from entering the critical section only if it is stuck in the while loop with the condition flag [j]

==true and turn=== j; this loop is the only one possible. If *Pi* is not ready to enter the critical section, then flag [j] ==false, and *P;* can enter its critical section. If *Pj* has set flag [j] to true and is also executing in its while statement, then either turn === i or turn === j. If turn == i, then *P;* will enter the critical section. If turn== j, then *Pi* will enter the critical section. However, once *Pi* exits its critical section, it will reset flag [j] to false, allowing *P;* to enter its critical section. If *Pi* resets flag [j] to true, it must also set turn to i. Thus, since *P;* does not change the value of the variable turn while executing the while statement, *P;* will enter the critical section (progress) after at most one entry by *P1* (bounded waiting).

## SYNCHRONIZATION HARDWARE

It can generally state that any solution to the critical-section problem requires a simple tool-a **lock.** Race conditions are prevented by requiring that critical regions be protected by locks. That is, a process must acquire a lock before entering a critical section; it releases the lock when it exits the critical section.

The critical-section problem could be solved simply in a single-processor environment if it could prevent interrupts from occurring while a shared variable was being modified. In this way, it could be sure that the current sequence of instructions would be allowed to execute in order without preemption. No other instructions would be run, so no unexpected modifications could be made to the shared variable. This is often the approach taken by nonpreemptive kernels.

Disabling interrupts on a multiprocessor can be time consuming, since the message is passed to all the processors. This message passing delays entry into each critical section, and system efficiency decreases.

Many modern computer systems therefore provide special hardware instructions that allow us either to test and modify the content of a word or to swap the contents of two words atomically.

- Definition:

```
boolean test_and_set (boolean *target)
  {
      boolean rv = *target;
      *target = TRUE; return rv:
  }
```

The definition of the test and set() instruction

- Solution:

```
do{
while (test_and_set(&lock))
      ; /* do nothing */
        /* critical section */ lock = false;
        /* remainder section */
```

```
} while (true);
```

Mutual-exclusion implementation with test and set().

The compare and swap() instruction, operates on three operands.

- Definition:

```
int compare _and_swap(int *value, int expected, int new_value) { int temp =
  *value;
  if (*value == expected)
    *value = new_value; return
temp;
}
```

The definition of the compare and swap() instruction.

- Solution:

```
do{
while (compare_and_swap(&lock, 0, 1) != 0)
    ; /* do nothing */
  /* critical section */
lock = 0;
  /* remainder section */
} while (true);
```

Mutual-exclusion implementation with the compare and swap() instruction

***Mutex Locks***

- Simplest tool s to solve the critical-section problem is **mutex lock.**
- A process must acquire the lock before entering a critical section; it releases the lock when it exits the critical section.
- The acquire()function acquires the lock, and the release() function releases the lock

➢ A mutex lock has a boolean variable **available** whose value indicates if the lock is available or not. If the lock is available, a call to acquire() succeeds, and the lock is then considered unavailable. A process that attempts to acquire an unavailable lock is blocked until the lock is released.

 Solution to the critical-section problem using mutex locks.

**acquire() {**

**while (!available)**

**; /* busy wait */ available = false;;**

**}**

**do**

{

*acquire lock*

critical section

*release lock*

remainder section

} while (TRUE);

 The definition of release() is

**release() {**

**available = true;**

**}**

- Calls to either acquire() or release() must be performed atomically.
- The main disadvantage of the implementation given here is that it requires busy waiting. While a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the call to acquire(). In fact, this type of mutex lock is also called a **spinlock** because the process "spins" while waiting for the lock to become available.

## SEMAPHORE

The hardware-based solutions to the critical-section problem presented are complicated for application programmers to use. To overcome this difficulty a synchronization tool called a semaphore is used. **A semaphore S is an integer variable that, apart from initialization, is accessed only through two standard atomic operations: wait () and signal ().** The wait () operation was originally termed P (from the Dutch *proberen*, "to test"); signal() was originally called V (from *verhogen*, "to increment"). The definition of wait () is as follows:

**wait(S)**

```
{
    while S <= 0   II busy waiting
s--;
}
```

The definition of signal() is as follows: signal(S)

```
{
S++;
}
```

All modifications to the integer value of the semaphore in the wait () and signal() operations must be executed indivisibly. That is, when one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value. In addition, in the case of wait (S), the testing of the integer value of S (S <- 0), as well as its possible modification (S--), must be executed without interruption.

***Semaphore Usage***

Operating systems often distinguish between counting and binary semaphores.

- The value of **a counting semaphore** can range over an unrestricted domain.
- The value of **a binary semaphore** can range only between 0 and 1.

Counting semaphores can be used to control access to a given resource consisting of a finite number of instances. The semaphore is initialized to the number of resources available. Each process that wishes to use a resource performs a wait() operation on the semaphore (thereby decrementing the count). When a process releases a resource, it performs a signal() operation (incrementing the count). When the count for the semaphore goes to 0, all resources are being used. After that, processes that wish to use a resource will block until the count becomes greater than 0.

- Consider $P_1$ and $P_2$ that require $S_1$ to happen before $S_2$

**Create a semaphore "synch" initialized to 0**

P1:

$S_1$;

**signal(synch);**

P2:

wait(synch); $S_2$;

**Semaphore Implementation**

When a process executes the wait() operation and finds that the semaphore value is not positive, it must wait. However, rather than engaging in busy waiting, the process can block itself. The block operation places a process into a waiting queue associated with the semaphore, and the state of the process is switched to the waiting state. Then control is transferred to the CPU scheduler, which selects another process to execute.

A process that is blocked, waiting on a semaphore S, should be restarted when some other process executes a signal() operation. The process is restarted by a wakeup() operation, which changes the process from the waiting state to the ready state. The process is then placed in the ready queue.

- a semaphore can be defined as follows:

```
typedef struct{ int
value;
struct process *list;

} semaphore;
```

- The wait() semaphore operation can be defined as

```
wait(semaphore *S) { S->value--;
if (S->value < 0) {
add this process to S->list; block();
}
}
```

signal() semaphore operation can be defined as

```
signal(semaphore *S)
{
S->value++;
if (S->value <= 0)


{
remove a process P from S->list; wakeup(P);
}
```

The block() operation suspends the process that invokes it. The wakeup(P) operation resumes the execution of a blocked process P.

If a semaphore value is negative, its magnitude is the number of processes waiting on that semaphore. This fact results from switching the order of the decrement and the test in the implementation of the wait() operation.

## Classical Problem on Synchronization:

There are various types of problem which are proposed for synchronization scheme such as

Bounded Buffer Problem: This problem was commonly used to illustrate the power of synchronization primitives. In this scheme we assumed that the pool consists of 'N' buffer and each capable of holding one item. The 'mutex' semaphore provides mutual exclusion for access to the buffer pool and is initialized to the value one. The empty and full semaphores count the number of empty and full buffer respectively. The semaphore empty is initialized to 'N' and the semaphore full is initialized to zero. This problem is known as procedure and consumer problem. The code of the producer is producing full buffer and the code of consumer is producing empty buffer. The structure of producer process is as follows:

```
do {

produce an item in nextp
```

. . . . . .

Wait (empty); Wait (mutex);

. . . . . .

add nextp to buffer

. . . . . .

Signal (mutex);

Signal (full);

} While (1);

The structure of consumer process is as follows:

do {

Wait (full); Wait (mutex);

. . . . . .

Remove an item from buffer to nextc

. . . . . .

Signal (mutex); Signal (empty);

. . . . . .

Consume the item in nextc;

. . . . . .. . .. .

} While (1);

Reader Writer Problem: In this type of problem there are two types of process are used such as Reader process and Writer process. The reader process is responsible for only reading and the writer process is responsible for writing. This is an important problem of synchronization which has several variations like The simplest one is referred as first reader writer problem which requires that no

reader will be kept waiting unless a writer has obtained permission to use the shared object. In other words no reader should wait for other reader to finish because a

writer is waiting.

o The second reader writer problem requires that once a writer is ready then the writer

performs its write operation as soon as possible. The structure of a reader process is as follows:

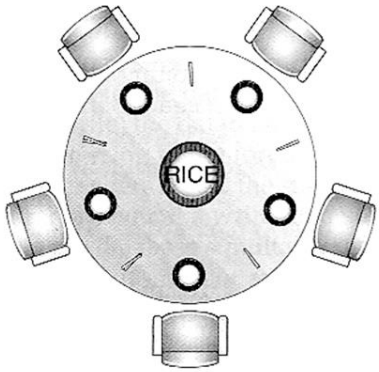
```
Wait (mutex);
Read count++;
if (read count == 1)
Wait (wrt);
Signal (mutex);
. . . . . .
Reading is performed
. . . . . .
Wait (mutex);
Read count --;
if (read count == 0)
Signal (wrt);
Signal (mutex);
```

The structure of the writer process is as follows: Wait (wrt);

```
Writing is performed;
Signal (wrt);
```

**Dining Philosopher Problem:**

Consider 5 philosophers to spend their lives in thinking & eating. A philosopher shares common circular table surrounded by 5 chairs each occupies by one philosopher. In the center of the table there is a bowl of rice and the table is laid with 6 chopsticks as shown in below figure.




When a philosopher thinks she does not interact with her colleagues. From time to time a philosopher gets hungry and tries to pickup two chopsticks that are closest to her. A philosopher may pickup one chopstick or two chopsticks at a time but she cannot pickup a chopstick that is already in hand of the neighbor. When a hungry philosopher has both her chopsticks at the same time, she eats without releasing her chopsticks. When she finished eating, she puts down both of her chopsticks and starts thinking again. This problem is considered as classic synchronization problem. According to this problem each chopstick is represented by a semaphore. A philosopher grabs the chopsticks by executing the wait operation on that semaphore. She releases the chopsticks by executing the signal operation on the appropriate semaphore. The structure of dining philosopher is as follows:

```
do{

Wait ( chopstick [i]);

Wait (chopstick [(i+1)%5]);

. . . . . .

Eat

. . . . . .

Signal (chopstick [i]);
```

Signal (chopstick [(i+1)%5]);

. . . . . .

Think

. . . . . .

} While (1);

**Monitor:**

It is characterized as a set of programmer defined operators. Its representation consists of declaring of variables, whose value defines the state of an instance. The syntax of monitor is as follows. Monitor monitor_name

{

Shared variable declarations Procedure body P1 (……) {

. . . . . .

}

Procedure body P2 (……) {

. . . .

}

. . .

Procedure body Pn (……) {

. . . . . .

}

{

Initialization Code

}

}

**Deadlock:**

In a multiprogramming environment several processes may compete for a finite number of resources. A process request resources; if the resource is available at that time a process enters the wait state. Waiting process may never change its state because the resources requested are held by other waiting process. This situation is known as deadlock.

**System Model:**

A system consists of a finite number of resources to be distributed among a number of competing processes. The resources are partitioned into several types each of which consists of a number of identical instances. A process may utilized a resources in the following sequence

•Request: In this state one can request a resource.

•Use: In this state the process operates on the resource.

•Release: In this state the process releases the resources.

**Deadlock Characteristics:**

In a deadlock process never finish executing and system resources are tied up. A deadlock situation can arise if the following four conditions hold simultaneously in a system.

•**Mutual Exclusion**: At a time only one process can use the resources. If another process requests that resource, requesting process must wait until the resource has been released.

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

•**No Preemption**: Resources allocated to a process can't be forcibly taken out from it unless it releases that resource after completing the task.

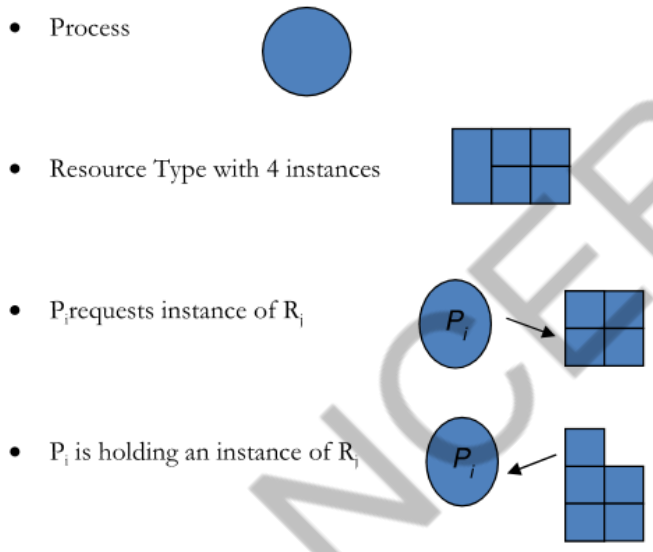
•**Circular Wait**: A set {P0, P1, …….Pn} of waiting state/ process must exists such that P0 is

waiting for a resource that is held by P1, P1 is waiting for the resource that is held by P2 ….. P(n – 1) is waiting for the resource that is held by Pn and Pn is waiting for the resources that is

held by P4.

**Resource Allocation Graph:**

Deadlock can be described more clearly by directed graph which is called system resource allocation graph. The graph consists of a set of vertices 'V' and a set of edges 'E'. The set of vertices 'V' is partitioned into two different types of nodes such as P = {P1, P2, …….Pn}, the set of all the active processes in the system and R = {R1, R2, …….Rm}, the set of all the resource type in the system. A directed edge from process Pi to resource type Rj is denoted by Pi → Rj. It signifies that process Pi is an instance of resource type Rj and waits for that resource. A directed edge from resource type Rj to the process Pi which signifies that an instance of resource type Rj has been allocated to process Pi. A directed edge Pi → Rj is called as request edge and Rj → Pi is called as assigned edge.



When a process Pi requests an instance of resource type Rj then a request edge is inserted as resource allocation graph. When this request can be fulfilled, the request edge is transformed to an assignment edge. When the process no longer needs access to the resource it releases the resource and as a result the assignment edge is deleted. The resource allocation graph shown in below figure has the following situation.

•The sets P, R, E

P = {P1, P2, P3}

R = {R1, R2, R3, R4}

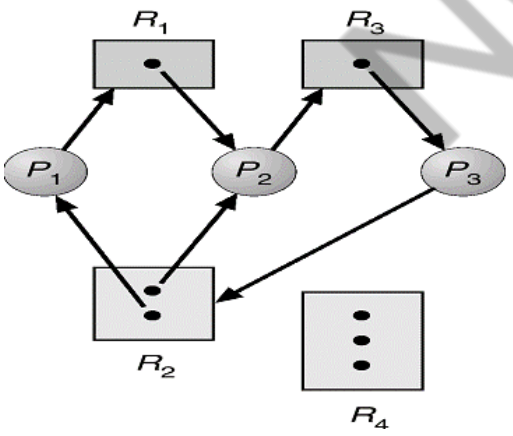
E = {P1 → R1,P2 → R3,R1 → P2,R2 → P2,R2 → P1,R3 → P3}

The resource instances are

Resource R1 has one instance

Resource R2 has two instances. Resource R3 has one instance

Resource R4 has three instances




The process states are:

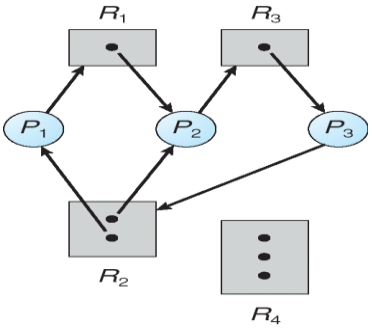
Process P1 is holding an instance of R2 and waiting for an instance of R1.

Process P2 is holding an instance of R1 and R2 and waiting for an instance R3.

Process P3 is holding an instance of R3.

The following example shows the resource allocation graph with a deadlock.

P1 -> R1 -> P2 -> R3 -> P3 -> R2 -> P1 P2 -> R3 -> P3 -> R2 -> P1

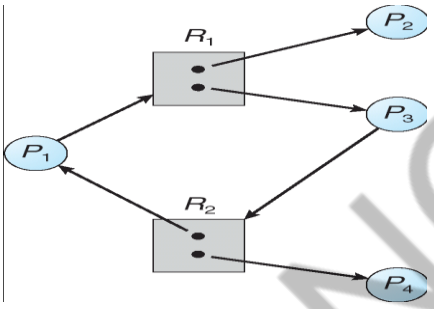



The following example shows the resource allocation graph with a cycle but no deadlock.

P1 -> R1 -> P3 -> R2 -> P1

No deadlock

P4 may release its instance of resource R2 Then it can be allocated to P3






## Methods for Handling Deadlocks

The problem of deadlock can deal with the following 3 ways.

We can use a protocol to prevent or avoid deadlock ensuring that the system will never enter

to a deadlock state.

We can allow the system to enter a deadlock state, detect it and recover. We can ignore the problem all together.

To ensure that deadlock never occur the system can use either a deadlock prevention or deadlock avoidance scheme.

## Deadlock Prevention:

Deadlock prevention is a set of methods for ensuring that at least one of these necessary conditions cannot hold.

**Mutual Exclusion**: The mutual exclusion condition holds for non sharable. The example is a

printer cannot be simultaneously shared by several processes. Sharable resources do not require mutual exclusive access and thus cannot be involved in a dead lock. The example is read only files which are in sharing condition. If several processes attempt to open the read

only file at the same time they can be guaranteed simultaneous access.

**Hold and wait**:To ensure that the hold and wait condition never occurs in the system, we

must guaranty that whenever a process requests a resource it does not hold any other resources. There are two protocols to handle these problems such as one protocol that can be used requires each process to request and be allocated all its resources before it begins execution. The other protocol allows a process to request resources only when the process has no resource. These protocols have two main disadvantages. First, resource utilization may be low, since many of the resources may be allocated but unused for a long period. Second, starvation is possible. A process that needs several popular resources may have to wait

**No Preemption**: To ensure that this condition does not hold, a protocol is used. If a process

is holding some resources and request another resource that cannot be immediately allocated to it. The preempted one added to a list of resources for which the process is waiting. The process will restart only when it can regain its old resources, as well as the new ones that it is requesting. Alternatively if a process requests some resources, we first check whether they are available. If they are, we allocate them. If they are not available, we check whether they are allocated to some other process that is waiting for additional resources. If so, we preempt the desired resources from the waiting process and allocate them to the requesting process. If the resources are not either available or held by a waiting process, the requesting process must

wait.

**Circular Wait**:We can ensure that this condition never holds by ordering of all resource type

and to require that each process requests resource in an increasing order of enumeration. Let R = {R1, R2, …….Rn}be the set of resource types. We assign to each resource type a unique integer number, which allows us to compare two resources and to determine whether one

precedes another in our ordering. Formally, we define a one to one function F: R → N, where

N is the set of natural numbers. For example, if the set of resource types R includes tape drives, disk drives and printers, then the function F might be defined as follows:

F (Tape Drive) = 1,

F (Disk Drive) = 5,

F (Printer) = 12.

We can now consider the following protocol to prevent deadlocks: Each process can request resources only in an increasing order of enumeration. That is, a process can initially request any number of instances of a resource type, say Ri. After that, the process can request instances of resource type Rj if and only if F (Rj) >

F (Ri). If several instances of the same resource type are needed, defined previously, a process that wants to use the tape drive and printer at the same time must first request the tape drive and then request the printer.

## Deadlock Avoidance

Requires additional information about how resources are to be used.Simplest and most useful model requires that each process declare the maximum number of resources of each type that it may need.The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition.Resource-allocation state is defined by the number of available and allocated resources, and the maximum demands of the processes.

### Safe State

When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state.Systems are in safe state if there exists a safe sequence of all process. A sequence <P1, P2, …, Pn> of ALL the processes is the system such that for each Pi, the resources that Pi can still request can be satisfied by currently available resources + resources held by all the Pj, withj <i.That is:

•If Pi resource needs are not immediately available, then Pi can wait until all Pjhave finished.

•When Pj is finished, Pi can obtain needed resources, execute, return allocated resources, and terminate.

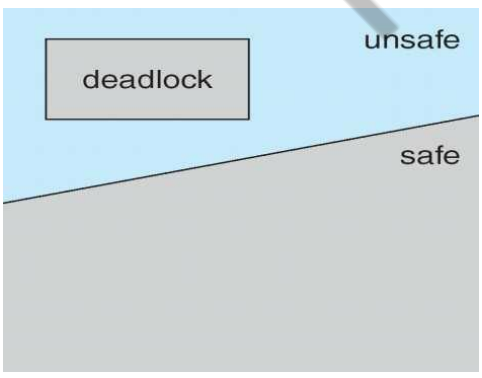
•When Pi terminates, Pi +1 can obtain its needed resources, and so on.

•If system is in safe state => No deadlock

If system in not in safe state => possibility of deadlock

•OS cannot prevent processes from requesting resources in a sequence that leads to deadlock

•Avoidance => ensue that system will never enter an unsafe state, prevent getting into deadlock



**Example**

| | Maximum Needs | Current Needs |
|---|---|---|
| $P_0$ | 10 | 5 |
| $P_1$ | 4 | 2 |
| $P_2$ | 9 | 2 |

suppose processes P0, P1, and P2 share 12 magnetic tape drives

•Currently 9 drives are held among the processes and 3 are available

•Question: Is this system currently in a safe state?

•Answer: Yes!

o Safe Sequence: <P1, P0, P2>

| | Maximum Needs | Current Needs |
|---|---|---|
| $P_0$ | 10 | 5 |
| $P_1$ | 4 | 2 |
| $P_2$ | 9 | 2 |

Suppose process P2 requests and is allocated 1 more tape drive.

•Question: Is the resulting state still safe?

•Answer: No! Because there does not exist a safe sequence anymore.

Only P1 can be allocated its maximum needs.

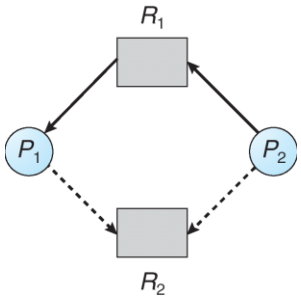
IFP0 and P2 request 5 more drives and 6 more drives, respectively, then the resulting

state will be deadlocked

**Resource Allocation Graph Algorithm**

In this graph a new type of edge has been introduced is known as claim edge. Claim edge Pi→Rj

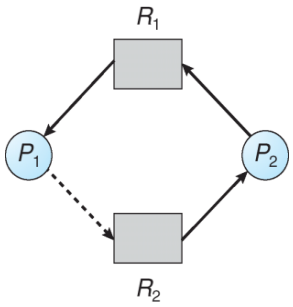
indicates that process Pj may request resource Rj; represented by a dashed line.Claim edge converts to request edge when a process requests a resource.Request edge converted to an assignment edge when the resource is allocated to the process.When a resource is released by a process, assignment edge reconverts to a claim edge.Resources must be claimed a priori in the system.




P2 requesting R1, but R1 is already allocated to P1.

Both processes have a claim on resource R2

What happens if P2 now requests resource R2?




Cannot allocate resource R2 to process P2 Why? Because resulting state is unsafe

•P1 could request R2, thereby creating deadlock!Use only when there is a single instance of each resource type

•Suppose that process Pi requests a resource Rj

•The request can be granted only if converting the request edge to an assignment edge does not result in the formation of a cycle in the resource allocation graph.

Here we check for safety by using cycle-detection algorithm.

## Banker's Algorithm

This algorithm can be used in banking system to ensure that the bank never allocates all its available cash such that it can no longer satisfy the needs of all its customer. This algorithm is applicable to a system with multiple instances of each resource type. When a new process enter in to the system it must declare the maximum number of instances of each resource type that it may need. This number may not exceed the total number of resources in the system. Several data structure must be maintained to implement the banker's algorithm.

Let,

•n = number of processes

•m = number of resources types

Available: Vector of length m. If Available[j] = k, there are k instances of resource type

Rj available.

Max: n x m matrix. If Max [i,j] = k, then process Pimay request at most k instances of resource

ype Rj.

Allocation: n x m matrix. If Allocation[i,j] = k then Pi is currently allocated k instances of Rj.
Need: n x m matrix. If Need[i,j] = k, then Pi may need k more instances of Rjto complete its

task.

Need [i,j] = Max[i,j] – Allocation [i,j].

**Safety Algorithm**

1.Let Workand Finish be vectors of length m and n, respectively. Initialize:

Work = Available

Finish [i] = false for i = 0, 1, …,n- 1.

2.Find and i such that both:

(a) Finish [i] = false

(b) Needi≤Work

If no such i exists, go to step 4.

3.Work = Work + Allocationi

Finish[i] = true go to step 2.

4.If Finish [i] == true for all i, then the system is in a safe state.

Resource Allocation Algorithm

Request = request vector for process Pi. If Requesti[j] = k then process Pi wants k instances of resource type Rj.

1.If Requesti≤Needigo to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim.

2.If Requesti≤Available, go to step 3. Otherwise Pi must wait, since resources are not available.

3.Pretend to allocate requested resources to Pi by modifying the state as follows:

Available = Available – Request; Allocationi= Allocationi + Requesti; Needi=Needi – Requesti;

•If safe ⇒ the resources are allocated to Pi.

•If unsafe ⇒ Pi must wait, and the old resource-allocation state is restored

Example

•5 processes P0 through P4;

•3 resource types:

A (10 instances), B (5instances), and C (7 instances).

Snapshot at time T0:

| | Allocation | Max | Available |
|---|---|---|---|
| | A B C | A B C | A B C |
| $P_0$ | 0 1 0 | 7 5 3 | 3 3 2 |
| $P_1$ | 2 0 0 | 3 2 2 | |
| $P_2$ | 3 0 2 | 9 0 2 | |
| $P_3$ | 2 1 1 | 2 2 2 | |
| $P_4$ | 0 0 2 | 4 3 3 | |

- The content of the matrix Need is defined to be Max – Allocation.

| | Need |
|---|---|
| | A B C |
| $P_0$ | 7 4 3 |
| $P_1$ | 1 2 2 |
| $P_2$ | 6 0 0 |
| $P_3$ | 0 1 1 |

$P_4$ 4 3 1

- The system is in a safe state since the sequence <$P_1$, $P_3$, $P_4$, $P_2$, $P_0$> satisfies safety criteria.

$P_1$ requests (1, 0, 2)

- Check that Request ≤ Available (that is, (1,0,2) ≤ (3,3,2) ⇒ true.

| Allocation | Need | Available |
|---|---|---|
| | A B C | A B C A B C |
| $P_0$ | 0 1 0 | 7 4 3 2 3 0 |
| $P_1$ | 3 0 2 | 0 2 0 |
| $P_2$ | 3 0 1 | 6 0 0 |
| $P_3$ | 2 1 1 | 0 1 1 |
| $P_4$ | 0 0 2 | 4 3 1 |

- Executing safety algorithm shows that sequence <$P_1$, $P_3$, $P_4$, $P_0$, $P_2$> satisfies safety requirement.
- Can request for (3,3,0) by $P_4$ be granted? –NO
- Can request for (0,2,0) by $P_0$ be granted? –NO (Results Unsafe)

## Deadlock Detection

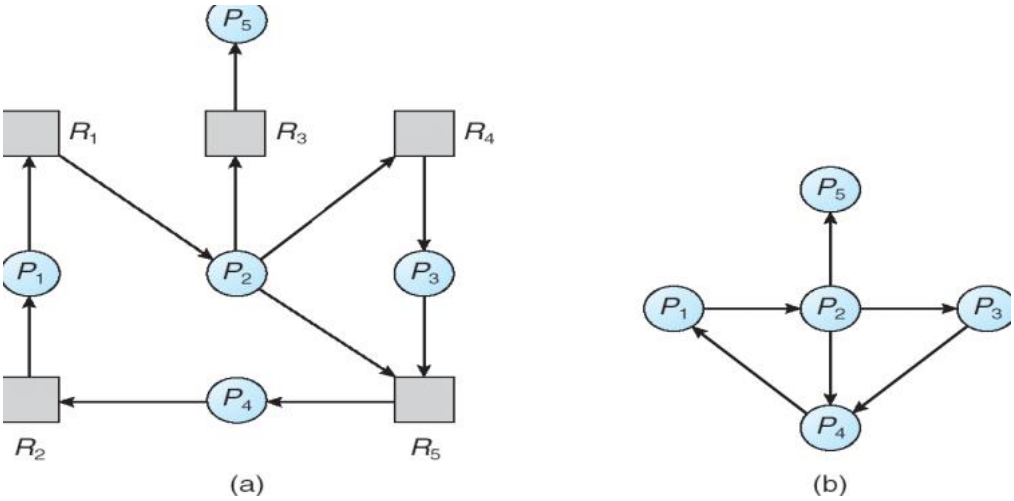
If a system doesn't employ either a deadlock prevention or deadlock avoidance, then deadlock situation may occur. In this environment the system must provide

•An algorithm to recover from the deadlock.

•An algorithm to remove the deadlock is applied either to a system which pertains single in

instance each resource type or a system which pertains several instances of a resource type. Single Instance of each Resource type

If all resources only a single instance then we can define a deadlock detection algorithm which uses a new form of resource allocation graph called "Wait for graph". We obtain this graph from the resource allocation graph by removing the nodes of type resource and collapsing the appropriate edges. The below figure describes the resource allocation graph and corresponding wait for graph




**Resource-Allocation Graph**

•For single instance

•Correspondin wait-for graph

Pi ->Pj(Pi is waiting for Pj to release a resource that Pi needs)

•Pi->Pj exist if and only if RAG contains 2 edges Pi ->Rq and Rq ->Pj for some resource Rq

**Several Instances of a Resource type**

The wait for graph scheme is not applicable to a resource allocation system with multiple instances of reach resource type. For this case the algorithm employs several data structures which are similar to those used in the banker's algorithm like available, allocation and request.

•Available: A vector of length m indicates the number of available resources of each type.

•Allocation: An n x m matrix defines the number of resources of each type currently allocated to each process.

•Request: An n x m matrix indicates the current request of each process. If Request [ij] = k, then process Pi is requesting k more instances of resource type. Rj.

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

(a) Work = Available

(b)For i = 1,2, …, n, if Allocationi≠ 0, then

Finish[i] = false;otherwise, Finish[i] = true.

2.Find an index i such that both:

a)Finish[i] == false

b)Requesti≤Work

If no such i exists, go to step 4.

3.Work = Work + Allocation

Finish [i] = true

Go to step 2

4.If Finish [i] = false, for some i, 1≤ i≤ n, then the system is in a deadlock state. Moreover, if Finish

[i] = false, then process Pi is deadlocked.

**Recovery from Deadlock**

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

**Process Termination:**

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

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

•Abort one process at a time until the deadlock cycle is eliminated:This method incurs

considerable overhead, since after each process is aborted, a deadlock detection algorithm

must be invoked to determine whether any processes are still deadlocked.

**Resource Preemption:**

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

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

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

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

# CST 206

# OPERATING SYSTEMS

# MODULE IV

**Memory Management**

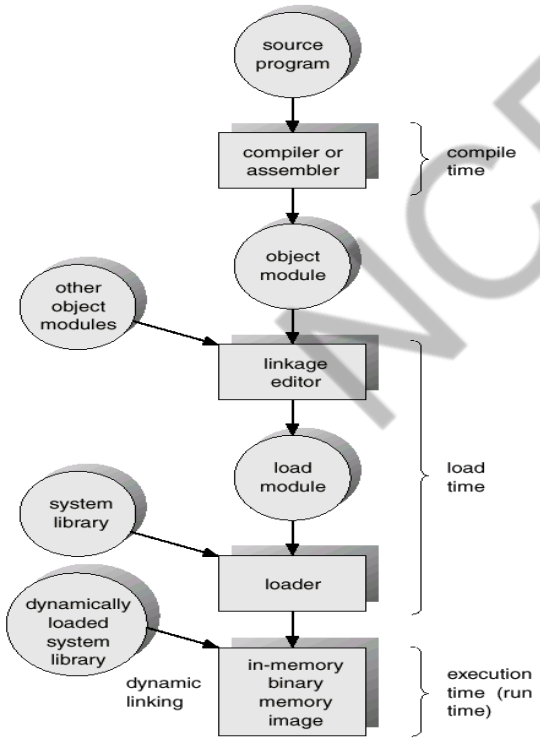
• Memory consists of a large array of words or bytes, each with its own address. The CPU fetches instructions from memory according to the value of the program counter. These instructions may cause additional loading from and storing to specific memory addresses.

• Memory unit sees only a stream of memory addresses. It does not know how they are generated.

• Program must be brought into memory and placed within a process for it to be run.

• Input queue – collection of processes on the disk that are waiting to be brought into memory for execution.






Address binding of instructions and data to memory addresses can happen at three different stages.

•Compile time: If memory location known a priori, absolute code can be generated; must recompile code if starting location changes.

Example: .COM-format programs in MS-DOS.

•Load time: Must generate relocatable code if memory location is not known at compile time.

•Execution time: Binding delayed until run time if the process can be moved during its execution from one memory segment to another. Need hardware support for address maps (e.g., relocation registers).

**Logical Versus Physical Address Space**

•The concept of a logical address space that is bound to a separate physical address space is central to proper memory management.

o Logical address – address generated by the CPU; also referred to as virtual address. o Physical address – address seen by the memory unit.

•The set of all logical addresses generated by a program is a logical address space; the set of all physical addresses corresponding to these logical addresses are a physical address space.

Address binding of instructions and data to memory addresses can happen at three different stages.

•Compile time: If memory location known a priori, absolute code can be generated; must recompile code if starting location changes.

Example: .COM-format programs in MS-DOS.

•Load time: Must generate relocatable code if memory location is not known at compile time.

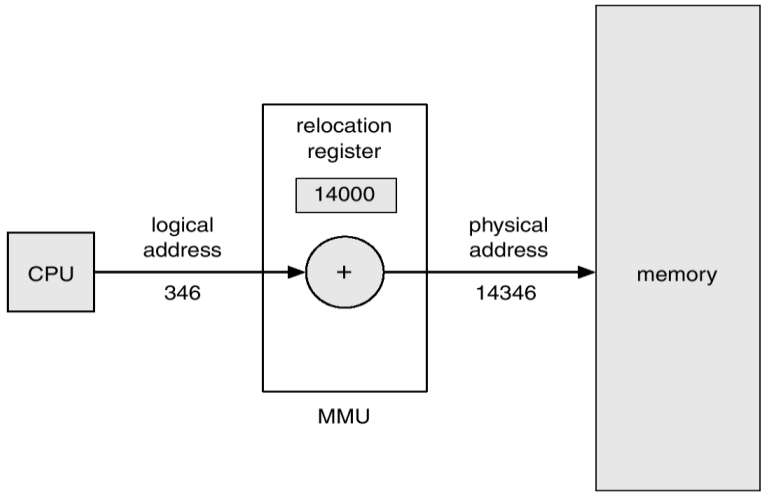
•Execution time: Binding delayed until run time if the process can be moved during its execution from one memory segment to another. Need hardware support for address maps (e.g., relocation registers).

Logical Versus Physical Address Space

•The concept of a logical address space that is bound to a separate physical address space is central to proper memory management.

o Logical address – address generated by the CPU; also referred to as virtual address. o Physical address – address seen by the memory unit.

•The set of all logical addresses generated by a program is a logical address space; the set of all physical addresses corresponding to these logical addresses are a physical address space.




This method requires hardware support slightly different from the hardware configuration. The base register is now called a relocation register. The value in the relocation register is added to every address generated by a user process at the time it is sent to memory.

•The user program never sees the real physical addresses. The program can create a pointer to location 346, store it in memory, manipulate it and compare it to other addresses. The user program deals with logical addresses. The memory mapping hardware converts logical addresses into physical addresses. The final location of a referenced memory address is not determined until the reference is made.

### Dynamic Loading

•Routine is not loaded until it is called.

•All routines are kept on disk in a relocatable load format.

•The main program is loaded into memory and is executed. When a routine needs to call another routine, the calling routine first checks to see whether the other the desired routine into memory and to update the program's address tables to reflect this change. Then control is passed to the newly loaded routine.

•Better memory-space utilization; unused routine is never loaded.

•Useful when large amounts of code are needed to handle infrequently occurring cases.

No special support from the operating system is required.

•Implemented through program design.

### Dynamic Linking

•Linking is postponed until execution time.

•Small piece of code, stub, is used to locate the appropriate memory-resident library routine, or to load the library if the routine is not already present.

•When this stub is executed, it checks to see whether the needed routine is already in memory. If not, the program loads the routine into memory.

•Stub replaces itself with the address of the routine, and executes the routine.

•Thus the next time that code segment is reached, the library routine is executed directly, incurring no cost for dynamic linking.

•Operating system is needed to check if routine is in processes' memory address.

•Dynamic linking is particularly useful for libraries.

## Swapping

•A process can be swapped temporarily out of memory to a backing store, and then brought back into memory for continued execution. For example, assume a multiprogramming environment with a round robin CPU scheduling algorithm. When a quantum expires, the memory manager will start to swap out the process that just finished, and to swap in another process to the memory space that has been freed. In the mean time, the CPU scheduler will allocate a time slice to some other process in memory. When each process finished its quantum, it will be swapped with another process. Ideally, the memory manager can swap processes fast enough that some processes will be in memory, ready to execute, when the CPU scheduler wants to reschedule the CPU. The quantum must also be sufficiently large that reasonable amounts of computing are done between swaps.

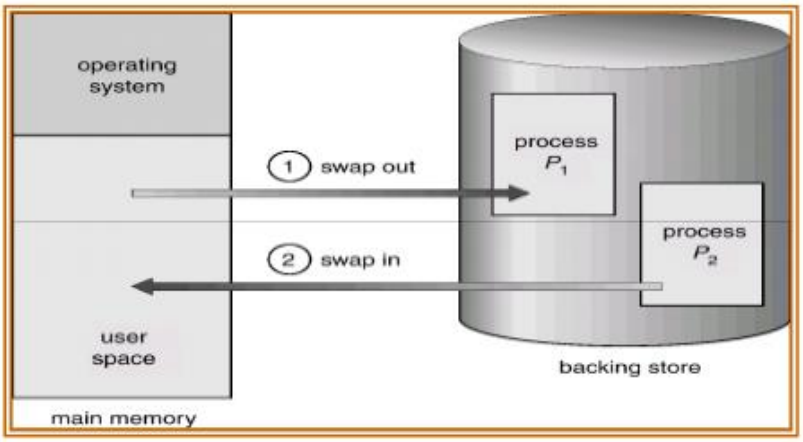
•Roll out, roll in – swapping variant used for priority-based scheduling algorithms. If a higher priority process arrives and wants service, the memory manager can swap out the lower priority process so that it can load and execute lower priority process can be swapped back in and continued. This variant is some times called roll out, roll in. Normally a process that is swapped out will be swapped back into the same memory space that it occupied previously. This restriction is dictated by the process cannot be moved to different locations. If execution time

binding is being used, then a process can be swapped into a different memory space, because the physical addresses are computed during execution time.

•Backing store – fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images. It must be large enough to accommodate copies of all memory images for all users, and it must provide direct access to these memory images. The system maintains a ready queue consisting of all processes whose memory images are scheduler decides to execute a process it calls the dispatcher. The dispatcher checks to see whether the next process in the queue is in memory. If not, and there is no free memory region, the dispatcher swaps out a process currently in memory and swaps in the desired process. It then reloads registers as normal and transfers control to the selected process.

•Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped.

•Modified versions of swapping are found on many systems (i.e., UNIX, Linux, and Windows)




**Contiguous Memory Allocation**

•Main memory is usually divided into two partitions:

o Resident operating system, usually held in low memory with interrupt vector. o User processes, held in high memory.

•In contiguous memory allocation, each process is contained in a single contiguous section of memory.

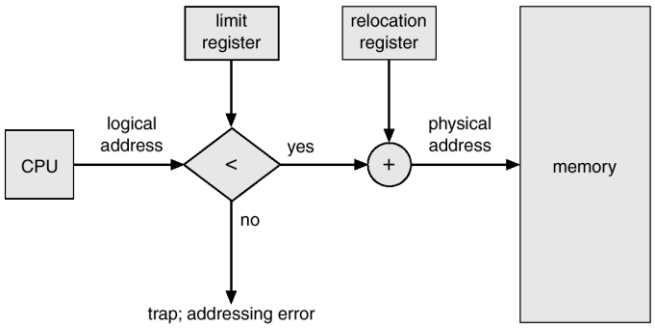
•Single-partition allocation

o Relocation-register scheme used to protect user processes from each other, and from

changing operating-system code and data

- Relocation register contains value of smallest physical address; limit register contains

range of logical addresses – each logical address must be less than the limit register.



**Multiple-partition allocation**

o Hole – block of available memory; holes of various size are scattered throughout

memory.

o When a process arrives, it is allocated memory from a hole large enough to

accommodate it.

o Operating system maintains information about:

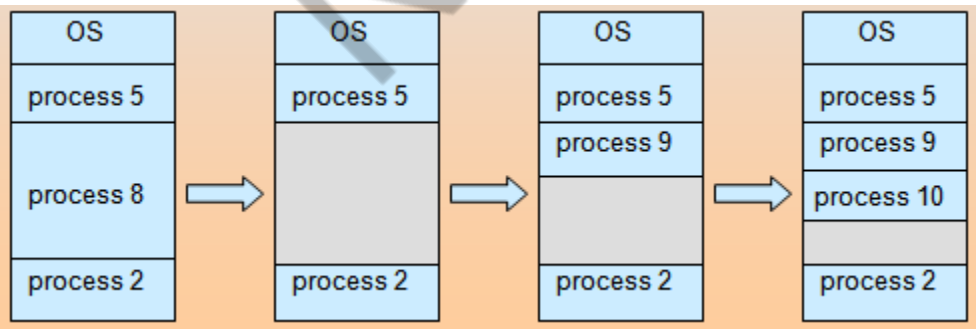
a)allocated partitions b) free partitions (hole)

o A set of holes of various sizes, is scattered throughout memory at any given time. When

a process arrives and needs memory, the system searches this set for a hole that is large enough for this process. If the hole is too large, it is split into two: one part is allocated to the arriving process; the other is returned to the set of holes. When a process terminates, it releases its block of memory, which is then placed back in the set of holes. If the new hold is adjacent to other holes, these adjacent holes are merged to form one larger hole.

o This procedure is a particular instance of the general dynamic storage allocation

problem, which is how to satisfy a request of size n from a list of free holes. There are many solutions to this problem. The set of holes is searched to determine which hole is best to allocate. The first-fit, best-fit and worst-fit strategies are the most common ones used to select a free hole from the set of available hole




- First-fit: Allocate the first hole that is big enough.

o Best-fit: Allocate the smallest hole that is big enough; must search entire list, unless

ordered by size.

o Worst-fit: Allocate the largest hole; must also search entire list.

Fragmentation

•External Fragmentation – total memory space exists to satisfy a request, but it is not contiguous.

•Internal Fragmentation – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used.

•Reduce external fragmentation by compaction

o Shuffle memory contents to place all free memory together in one large block.

o Compaction is possible only if relocation is dynamic, and is done at execution time.

**Paging**

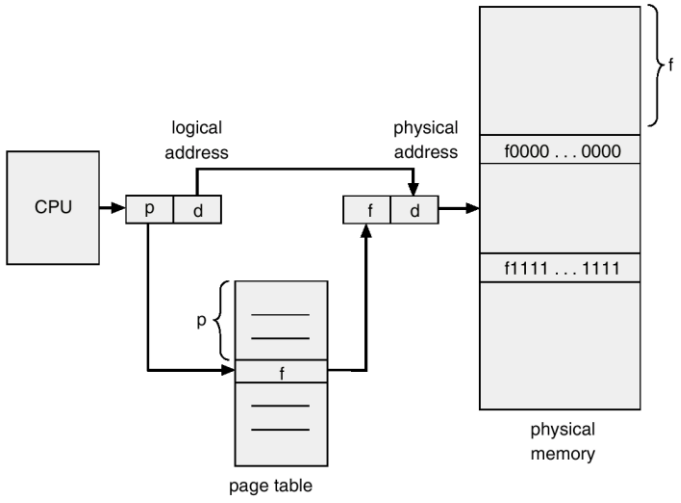
•Paging is a memory management scheme that permits the physical address space of a process to be non contiguous.

•Divide physical memory into fixed-sized blocks called frames (size is power of 2, for example 512 bytes).

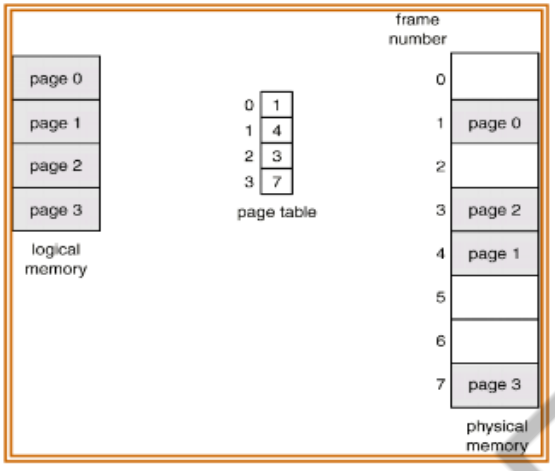
•Divide logical memory into blocks of same size called pages. When a process is to be executed, its pages are loaded into any available memory frames from the backing store. The backing store is divided into fixed sized blocks that are of the same size as the memory frames.

•The hardware support for paging is illustrated in below figure.

•Every address generated by the CPU is divided into two parts: a page number (p) and a page offset (d). The page number is used as an index into a page table. The page table contains the base address of each page in physical memory. This base address is combined with the page offset to define the physical memory address that is sent to the memory unit.



o The paging model of memory is shown in below figure. The page size is defined by the hardware. The size of a page is typically of a power of 2, varying between 512 bytes and 16 MB per page, depending on the computer architecture. The selection of a power of 2 as a page size makes the translation of a logical address into a page number and page offset particularly easy. If the size of logical address is 2m, and a page size is 2n addressing units, then the high order m-n bits of a logical address designate the page number, and the n low order bits designate the page offset.




- Keep track of all free frames.

•To run a program of size n pages, need to find n free frames and load program.

•Set up a page table to translate logical to physical addresses.

•Internal fragmentation may occur

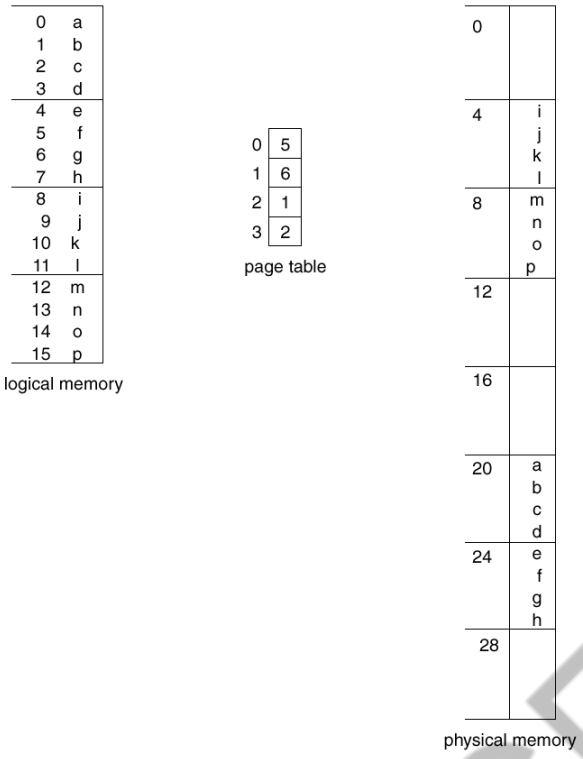
Let us take an example. Suppose a program needs 32 KB memory for allocation. The whole program is divided into smaller units assuming 4 KB and is assigned some address. The address consists of two parts such as:

•A large number in higher order positions and

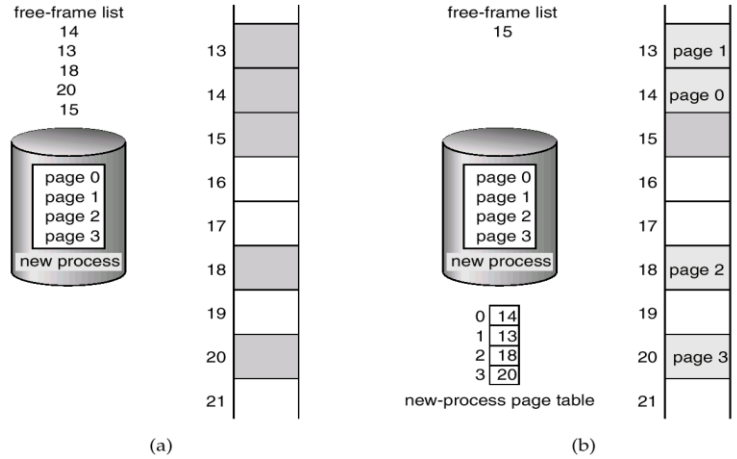
•Displacement or offset in the lower order bits.

The numbers allocated to pages are typically in power of 2 to simplify extraction of page numbers and offsets. To access a piece of data at a given address, the system first extracts the page number and the offset. Then it translates the page number to physical page frame and access data at offset in physical page frame. At this moment, the translation of the address by the OS is done using a page table. Page table is a linear array indexed by virtual page number which provides the physical page frame that contains the particular page. It employs a lookup process that extracts the page number and the offset. The system in addition checks that the page number is within the address space of process and retrieves the page number in the page table. Physical address will calculated by using the formula.

Physical address = page size of logical memory X frame number + offset




When a process arrives in the system to be executed, its size expressed in pages is examined. Each page of the process needs one frame. Thus if the process requires n pages, at least n frames must be available in memory. If n frames are available, they are allocated to this arriving process. The first page of the process is loaded into one of the allocated frames, and the frame number is put in the page table for this process. The next page is loaded into another frame, and its frame number is put into the page table and so on as in below figure. An important aspect of paging is the clear separation between the user's view of memory and the actual physical memory. The user program views that memory as one single contiguous space, containing only this one program. In fact, the user program is scattered throughout physical memory, which also holds other programs. The difference between the user's view of memory and the actual physical memory is reconciled by the address-translation hardware. The logical addresses are translated into physical addresses. This mapping is hidden from the user and is controlled by the operating system.




**Implementation of Page Table**

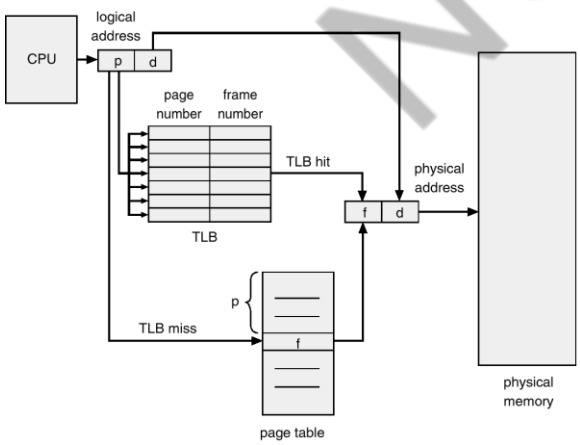
•Page table is kept in main memory.

•Page-tablebase register (PTBR) points to the page table.

•In this scheme every data/instruction-byte access requires two memory accesses. One for the page-table entry and one for the byte.

•The two memory access problem can be solved by the use of a special fast-lookup hardware cache called associative registers or associative memory or translation look-aside buffers(TLBs).

•Typically, the number of entries in a TLB is between 32 and 1024.



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

•If the page number is not in the TLB (known as a TLB miss), a memory reference to the

page table must be made. When the frame number is obtained, we can use it to access

memory

**Hit Ratio**

•Hit Ratio: the percentage of times that a page number is found in the associative registers.

•For example, if it takes 20 nanoseconds to search the associative memory and 100 nanoseconds to access memory; for a 98-percent hit ratio, we have

Effective memory-access time = 0.98 x 120 + 0.02 x 220

= 122 nanoseconds.

•The Intel 80486 CPU has 32 associative registers, and claims a 98-percent hit ratio.

Valid or invalid bit in a page table

•Memory protection implemented by associating protection bit with each frame.

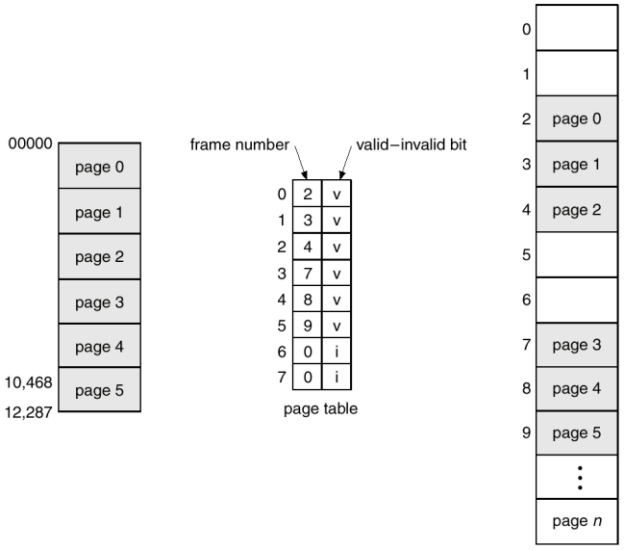
•Valid-invalid bit attached to each entry in the page table:

o "Valid" indicates that the associated page is in the process' logical address space, and is

thus a legal page.

o "Invalid" indicates that the page is not in the process' logical address space.

Pay attention to the following figure. The program extends to only address 10,468, any reference beyond that address is illegal. However, references to page 5 are classified as valid, so accesses to addresses up to 12,287 are valid. This reflects the internal fragmentation of paging.




**Structure of the Page Table**

Hierarchical Paging:

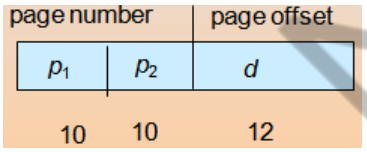
•A logical address (on 32-bit machine with 4K page size) is divided into:

o A page number consisting of 20 bits. o A page offset consisting of 12 bits.

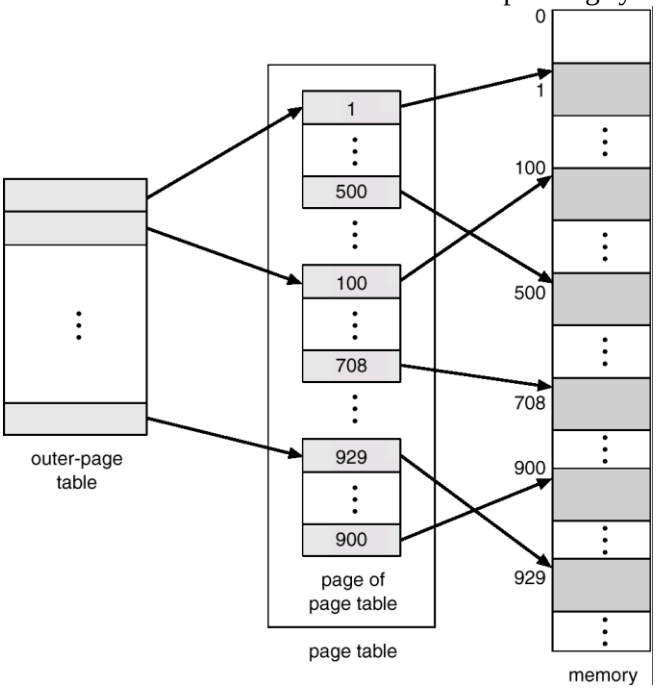
•Since the page table is paged, the page number is further divided into:

o A 10-bit page number. o A 10-bit page offset.

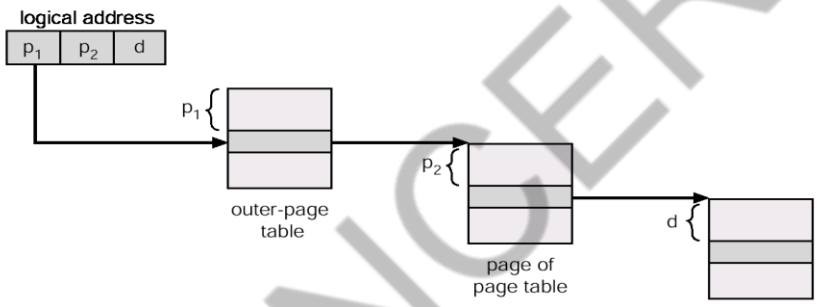
•Thus, a logical address is as follows




Where p1 is an index into the outer page table, and p2 is the displacement within the page of the outer page table.The below figure shows a two level page table scheme.



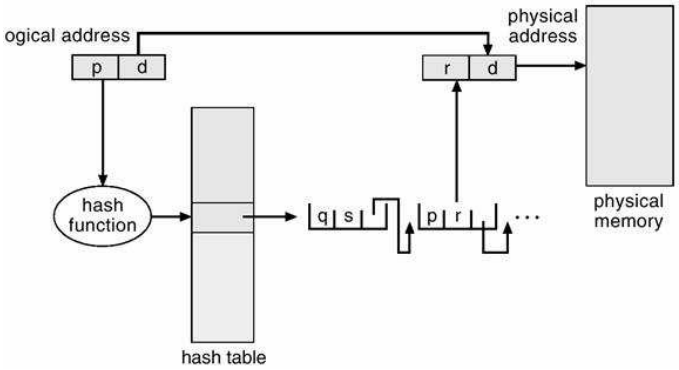

Address-translation scheme for a two-level 32-bit paging architecture is shown in below figure.






**Hashed Page Table:**

A common approach for handling address spaces larger than 32 bits is to use a hashed page table, with the hash value being the virtual page number. Each entry in the hash table contains a linked list of elements that has to the same location. Each element consists of three fields: (a) the virtual page number, (b) the value of the mapped page frame, and (c) a pointer to the next element in the linked list. The algorithm works as follows: The virtual page number in the virtual address is hashed into the hash table. The virtual page number is compared to field (a) in the first element in the linked list. If there is a match, the corresponding page frame (field (b)) is used to form the desired physical address. If there is no match, subsequent entries in the linked list are searched for a matching virtual page number. The scheme is shown in below figure.




**Inverted Page Table:**

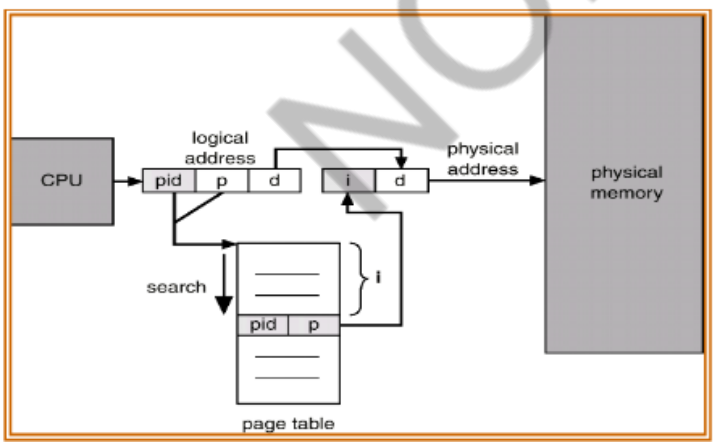
•One entry for each real page (frame) of memory.

•Entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page.

•There is only one page table in the system. Not per process.

•Decreases memory needed to store each page table, but increases time needed to search the table when a page reference occurs.

•Use hash table to limit the search to one — or at most a few — page-table entries




Each virtual address in the system consists of a triple <process-id, page-number, offset>. Each inverted page table entry is a pair <process-id, page-number> where the process-id assumes the role of the address space identifier. When a memory reference occurs, part of the virtual address, consisting of <process-id, page-number>, is presented to the memory subsystem. The inverted page table is then searched for a match. If a match is found say at entry i, then the physical address <i, offset> is generated. If no match is found, then an illegal address access has been attempted.

**Segmentation**

•Memory-management scheme that supports user view of memory.

•A program is a collection of segments. A segment is a logical unit such as:

Main program,

Procedure,

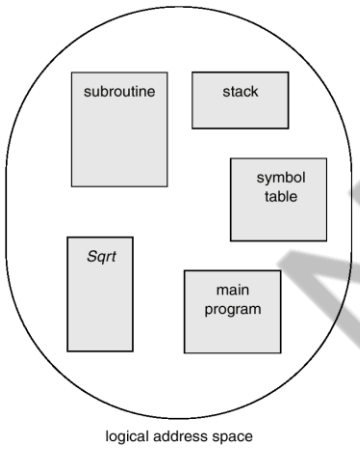
Function,

Method,

Object,

Local variables, global variables, Common block,

Stack, symbol table and arrays



logical address space



Segmentation is a memory management scheme that supports this user view of memory.

•A logical address space is a collection of segments. Each segment has a name and a length.

•The addresses specify both the segment name and the offset within the segment.

•The user therefore specifies each address by two quantities such as segment name and an offset. For simplicity of implementation, segments are numbered and are referred to by a segment number, rather than by a segment name.

•Logical address consists of a two tuples:

<segment-number, offset>

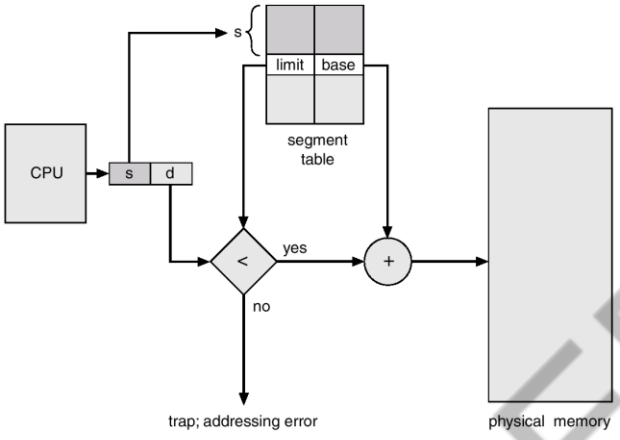
•Segment table – maps two-dimensional physical addresses; each table entry has:

o Base – contains the starting physical address where the segments reside in memory. o Limit – specifies the length of the segment.

•Segment-table base register (STBR) points to the segment table's location in memory.

•Segment-table length register (STLR) indicates number of segments used by a program;

Segment number s is legal if s< STLR




When the user program is compiled by the compiler it constructs the segments.

•The loader takes all the segments and assigned the segment numbers.

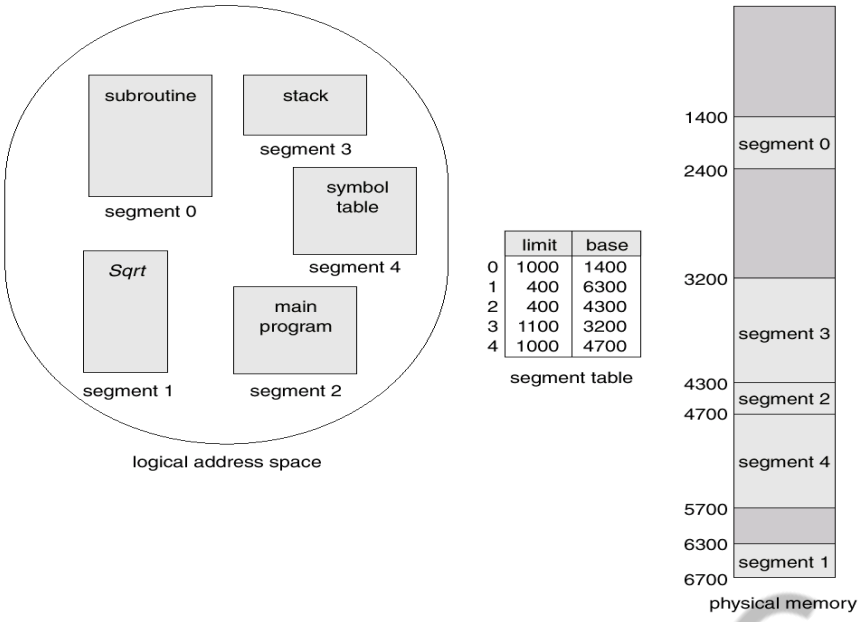
•The mapping between the logical and physical address using the segmentation technique is shown in above figure.

•Each entry in the segment table as limit and base address.

•The base address contains the starting physical address of a segment where the limit address specifies the length of the segment.

•The logical address consists of 2 parts such as segment number and offset.

•The segment number is used as an index into the segment table. Consider the below example is given below.




**Segmentation with Paging**

•Both paging and segmentation have advantages and disadvantages, that's why we can combine these two methods to improve this technique for memory allocation.

•These combinations are best illustrated by architecture of Intel-386.

•The IBM OS/2 is an operating system of the Intel-386 architecture. In this technique both segment table and page table is required.

•The program consists of various segments given by the segment table where the segment table contains different entries one for each segment.

•Then each segment is divided into a number of pages of equal size whose information is maintained in a separate page table.

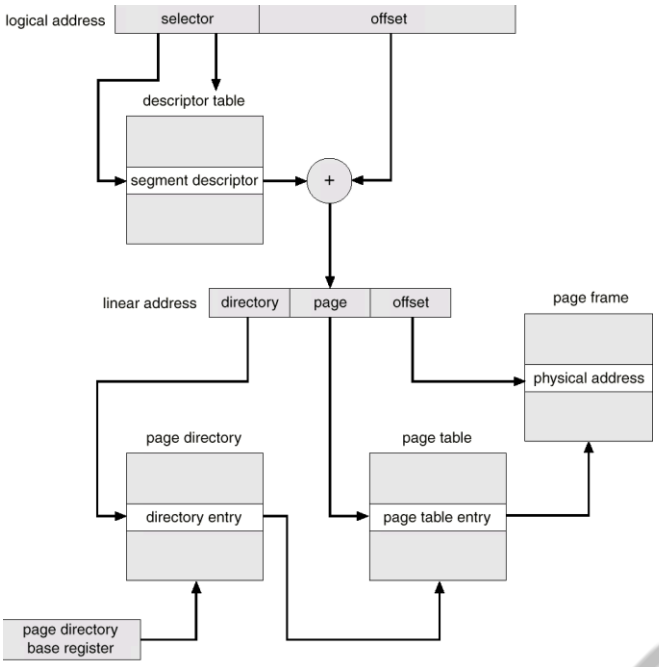
If a process has four segments that is 0 to 3 then there will be 4 page tables for that process, one for each segment.

•The size fixed in segmentation table (SMT) gives the total number of pages and therefore maximum page number in that segment with starting from 0.

•If the page table or page map table for a segment has entries for page 0 to 5.

•The address of the entry in the PMT for the desired page p in a given segment s can be obtained by B + P where B can be obtained from the entry in the segmentation table.

•Using the address (B +P) as an index in page map table (page table), the page frame (f) can be obtained and physical address can be obtained by adding offset to page frame.




**Virtual Memory**

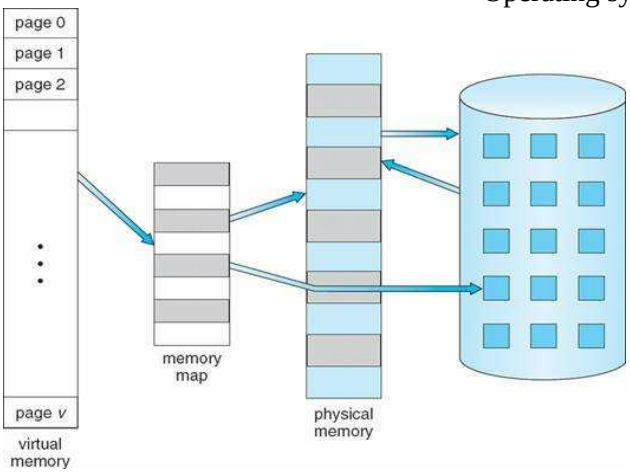
•It is a technique which allows execution of process that may not be compiled within the primary memory.

•It separates the user logical memory from the physical memory. This separation allows an extremely large memory to be provided for program when only a small physical memory is available.

•Virtual memory makes the task of programming much easier because the programmer no longer needs to working about the amount of the physical memory is available or not.

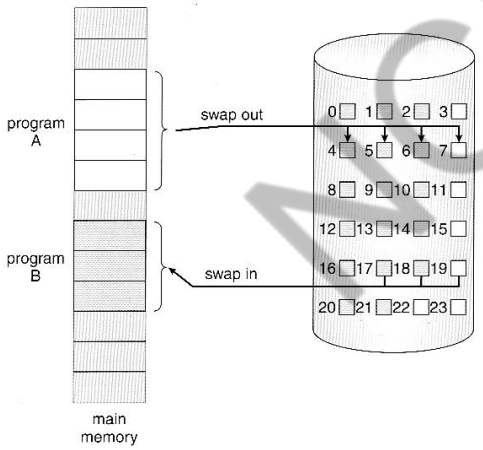
•The virtual memory allows files and memory to be shared by different processes by page sharing.

•It is most commonly implemented by demand paging.

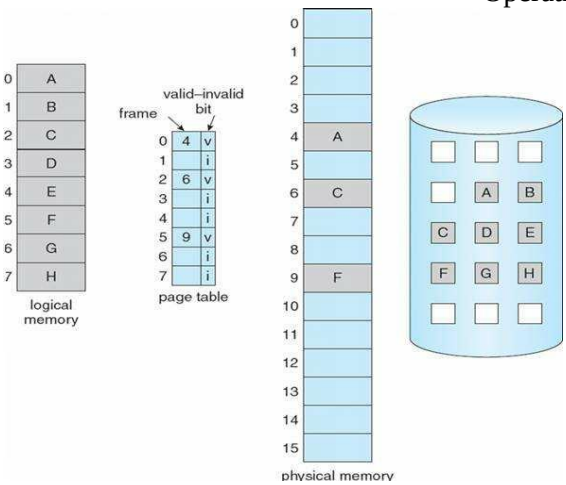



**Demand Paging**

A demand paging system is similar to the paging system with swapping feature. When we want to execute a process we swap it into the memory. A swapper manipulates entire process where as a pager is concerned with the individual pages of a process. The demand paging concept is using pager rather than swapper. When a process is to be swapped in, the pager guesses which pages will be used before the process is swapped out again. Instead of swapping in a whole process, the pager brings only those necessary pages into memory. The transfer of a paged memory to contiguous disk space is shown in below figure.




Thus it avoids reading into memory pages that will not used any way decreasing the swap time and the amount of physical memory needed. In this technique we need some hardware support to distinct between the pages that are in memory and those that are on the disk. A valid and invalid bit is used for this purpose. When this bit is set to valid it indicates that the associate page is in memory. If the bit is set to invalid it indicates that the page is either not valid or is valid but currently not in the disk




Marking a page invalid will have no effect if the process never attempts to access that page. So while a process executes and access pages that are memory resident, execution proceeds normally. Access to a page marked invalid causes a page fault trap. It is the result of the OS's failure to bring the desired page into memory.

Procedure to handle page fault

If a process refers to a page that is not in physical memory then

•We check an internal table (page table) for this process to determine whether the reference was valid or invalid.

•If the reference was invalid, we terminate the process, if it was valid but not yet brought in, we have to bring that from main memory.

•Now we find a free frame in memory.

•Then we read the desired page into the newly allocated frame.

•When the disk read is complete, we modify the internal table to indicate that the page is now in memory.

•We restart the instruction that was interrupted by the illegal address trap. Now the process can access the page as if it had always been in memory.

**Page Replacement**

•Each process is allocated frames (memory) which hold the process's pages (data)

•Frames are filled with pages as needed – this is called demand paging

Over-allocation of memory is prevented by modifying the page-fault service routine to replace pages

•The job of the page replacement algorithm is to decide which page gets victimized to make room for a new page

•Page replacement completes separation of logical and physical memory

Page Replacement Algorithm

Optimal algorithm

•Ideally we want to select an algorithm with the lowest page-fault rate

•Such an algorithm exists, and is called (unsurprisingly) the optimal algorithm:

•Procedure: replace the page that will not be used for the longest time (or at all) – i.e. replace the page with the greatest forward distance in the reference string

•Example using 4 frames:

| **Reference #** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Page referenced** | **1** | **2** | **3** | **4** | **1** | **2** | **5** | **1** | **2** | **3** | **4** | **5** |
| **Frames** | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 4 | 4 |
| _ = faulting page | | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| | | | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 |
| | | | | 4 | 4 | 4 | 5 | 5 | 5 | 5 | 5 | 5 |

- Analysis: 12 page references, 6 page faults, 2 page replacements. Page faults per number of frames = 6/4 = 1.5
- Unfortunately, the optimal algorithm requires special hardware (crystal ball, magic mirror, etc.) not typically found on today's computers
- Optimal algorithm is still used as a metric for judging other page replacement algorithms

**FIFO algorithm**

- Replaces pages based on their order of arrival: oldest page is replaced
- Example using 4 frames:

| **Reference #** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Page referenced** | **1** | **2** | **3** | **4** | **1** | **2** | **5** | **1** | **2** | **3** | **4** | **5** |
| **Frames** | 1 | 1 | 1 | 1 | 1 | 1 | 5 | 5 | 5 | 5 | 4 | 4 |
| _ = faulting page | | 2 | 2 | 2 | 2 | 2 | 2 | 1 | 1 | 1 | 1 | 5 |
| | | | 3 | 3 | 3 | 3 | 3 | 3 | 2 | 2 | 2 | 2 |
| | | | | 4 | 4 | 4 | 4 | 4 | 4 | 3 | 3 | 3 |

- Analysis: 12 page references, 10 page faults, 6 page replacements. Page faults per number of frames = 10/4 = 2.5

**LFU algorithm (page-based)**

•procedure: replace the page which has been referenced least often

•For each page in the reference string, we need to keep a reference count. All reference counts start at 0 and are incremented every time a page is referenced.

•example using 4 frames:

| Reference # | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Page referenced | 1 | 2 | 3 | 4 | 1 | 2 | 5 | 1 | 2 | 3 | 4 | 5 |
| Frames | $^{1}\underline{1}$ | $^{1}1$ | $^{1}1$ | $^{1}1$ | $^{2}1$ | $^{2}1$ | $^{2}1$ | $^{3}1$ | $^{3}1$ | $^{3}1$ | $^{3}1$ | $^{3}1$ |
| _ = faulting page | | $^{1}\underline{2}$ | $^{1}2$ | $^{1}2$ | $^{1}2$ | $^{2}2$ | $^{2}2$ | $^{2}2$ | $^{3}2$ | $^{3}2$ | $^{3}2$ | $^{3}2$ |
| $^{n}$ = reference count | | | $^{1}\underline{3}$ | $^{1}3$ | $^{1}3$ | $^{1}3$ | $^{1}\underline{5}$ | $^{1}5$ | $^{1}5$ | $^{2}\underline{3}$ | $^{2}3$ | $^{2}\underline{5}$ |
| | | | | $^{1}\underline{4}$ | $^{1}4$ | $^{1}4$ | $^{1}4$ | $^{1}4$ | $^{1}4$ | $^{1}4$ | $^{2}4$ | $^{2}4$ |

At the 7th page in the reference string, we need to select a page to be victimized. Either 3 or 4 will do since they have the same reference count (1). Let's pick 3.

•Likewise at the 10th page reference; pages 4 and 5 have been referenced once each. Let's pick page 4 to victimize. Page 3 is brought in, and its reference count (which was 1 before we paged it out a while ago) is updated to 2.

•Analysis: 12 page references, 7 page faults, 3 page replacements. Page faults per number of frames = 7/4 = 1.75

**LFU algorithm (frame-based)**

•Procedure: replace the page in the frame which has been referenced least often

•Need to keep a reference count for each frame which is initialized to 1 when the page is paged in, incremented every time the page in the frame is referenced, and reset every time the page in the frame is replaced

•Example using 4 frames

| Reference # | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Page referenced | 1 | 2 | 3 | 4 | 1 | 2 | 5 | 1 | 2 | 3 | 4 | 5 |
| Frames | [1]1 | [1]1 | [1]1 | [1]1 | [2]1 | [2]1 | [2]1 | [3]1 | [3]1 | [3]1 | [3]1 | [3]1 |
| _ = faulting page | | [1]2 | [1]2 | [1]2 | [1]2 | [2]2 | [2]2 | [2]2 | [3]2 | [3]2 | [3]2 | [3]2 |
| [n] = reference count | | | [1]3 | [1]3 | [1]3 | [1]3 | [1]5 | [1]5 | [1]5 | [1]3 | [1]3 | [1]5 |
| | | | | [1]4 | [1]4 | [1]4 | [1]4 | [1]4 | [1]4 | [1]4 | [2]4 | [2]4 |

At the 7th reference, we victimize the page in the frame which has been referenced least often -- in this case, pages 3 and 4 (contained within frames 3 and 4) are candidates, each with a reference count of 1. Let's pick the page in frame 3. Page 5 is paged in and frame 3's reference count is reset to 1.

•At the 10th reference, we again have a page fault. Pages 5 and 4 (contained within frames 3 and 4)are candidates, each with a count of 1. Let's pick page 4. Page 3 is paged into frame 3, and frame 3's reference count is reset to 1.

Analysis: 12 page references, 7 page faults, 3 page replacements. Page faults per number of frames = 7/4 = 1.75

**LRU algorithm**

•Replaces pages based on their most recent reference – replace the page with the greatest backward distance in the reference string

•Example using 4 frames:

| Reference # | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|

| Page referenced | 1 | 2 | 3 | 4 | 1 | 2 | 5 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Frames | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 5 |
| _ = faulting page | | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| | | | 3 | 3 | 3 | 3 | 5 | 5 | 5 | 5 | 4 | 4 |
| | | | | 4 | 4 | 4 | 4 | 4 | 4 | 3 | 3 | 3 |

Analysis: 12 page references, 8 page faults, 4 page replacements. Page faults per number of frames = 8/4 = 2

•One possible implementation (not necessarily the best):

o Every frame has a time field; every time a page is referenced, copy the current time into its frame's time field

o When a page needs to be replaced, look at the time stamps to find the oldest

**Thrashing**

•If a process does not have "enough" pages, the page-fault rate is very high

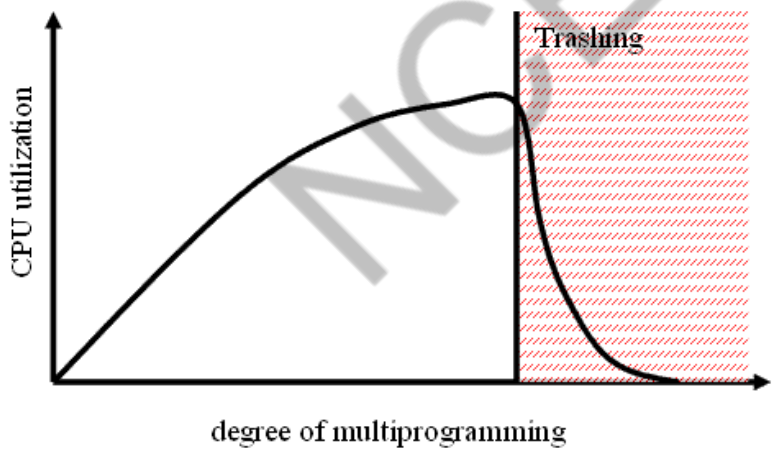
–low CPU utilization

–OS thinks it needs increased multiprogramming

–adds another process to system

•Thrashing is when a process is busy swapping pages in and out

•Thrashing results in severe performance problems. Consider the following scenario, which is based on the actual behaviour of early paging systems. The operating system monitors CPU utilization. If CPU utilization is too low, we increase the degree of multiprogramming by introducing a new process to the system. A global page replacement algorithm is used; it replaces pages with no regard to the process to which they belong. Now suppose that a process enters a new phase in its execution and needs more frames.



# CST 206

# OPERATING SYSTEMS

# MODULE V

## 5.1 Storage Management

This session gives an overview of the physical structure of secondary and tertiary storage devices.

### Magnetic Disks

- Magnetic disks provide the bulk of secondary storage for modern computer systems. Conceptually, disks are relatively simple each disk platter has a flat circular shape, like a CD. Common platter diameters range from1.8 to 3.5 inches. The two surfaces of a platter are covered with a magnetic material. We store information by recording it magnetically on the platters.
- A read–write head "flies" just above each surface of every platter. The heads are attached to a disk arm that moves all the heads as a unit. The surface of a platter is logically divided into circular tracks, which are subdivided into sectors. The set of tracks that are at one arm position makes up a cylinder. There may be thousands of concentric cylinders in a disk drive, and each track may contain hundreds of sectors. The storage capacity of common disk drives is measured in gigabytes.
- When the disk is in use, a drive motor spins it at high speed. Most drives rotate 60 to 250 times per second, specified in terms of rotations per minute (RPM). Disk speed has two parts. The transfer rate is the rate at which data flow between the drive and the computer. The positioning time, or random-access time, consists of two parts: the time necessary to move the disk arm to the desired cylinder, called the seek time, and the time necessary for the desired sector to rotate to the disk head, called the rotational latency.

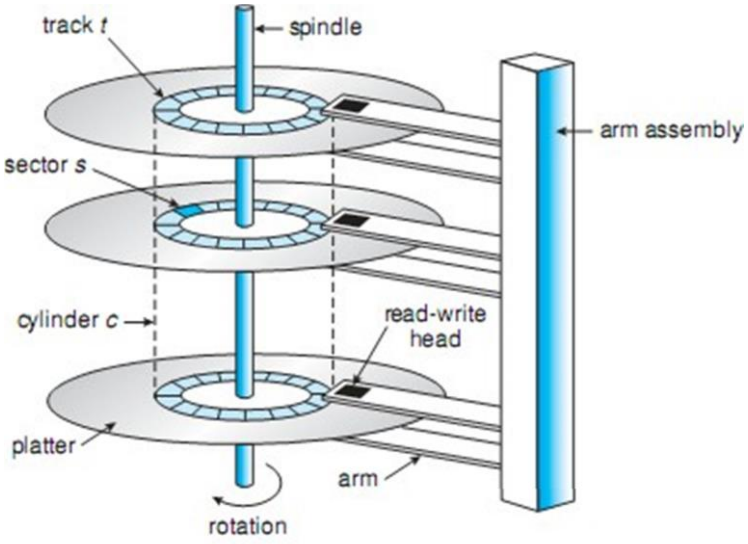



**Figure 10.1** Moving-head disk mechanism.

- A disk drive is attached to a computer by a set of wires called an I/O bus. Several kinds of buses are available, including advanced technology attachment (ATA), serial ATA (SATA), e-SATA, universal serial bus (USB), and fiber channel (FC). The data transfers on a bus are carried out by special electronic processors called controllers.

## Disk Structure

- Modern magnetic disk drives are addressed as large one-dimensional arrays of logical blocks, where the logical block is the smallest unit of transfer. The size of a logical block is usually 512 bytes, although some disks can be low-level formatted to have a different logical block size, such as 1,024 bytes.

- The one-dimensional array of logical blocks is mapped onto the sectors of the disk sequentially. Sector 0 is the first sector of the first track on the outermost cylinder. The mapping proceeds in order through that track, then through the rest of the tracks in that cylinder, and then through the rest of the cylinders from outermost to innermost. By using this mapping, we an—at least in theory—convert a logical block number into an old-style disk address that consists of a cylinder number, a track number within that cylinder, and a sector number within that track.
- In practice it is difficult to perform this translation because of the following reasons
    1. most disks have some defective sectors, but the mapping hides this by substituting spare sectors from elsewhere on the disk.

2. Second, the number of sectors per track is not a constant on some drives.

- Arranging tracks in different manner , which includes

1. **Constant Linear Velocity (CLV):** The density of bits per track is uniform. The farther a track is from the center of the disk, the greater its length, so the more sectors it can hold. As we move from outer zones to inner zones, the number of sectors per trackdecreases.
2. **Constant Angular Velocity (CAV):** Tracks in the outermost zone typically hold 40 percent more sectors than tracks in the innermost zone. The drive increases its rotation speed as the head moves from the outer to the inner tracks to keep the same rate of data moving under the head. This method is used in CD-ROM and DVD-ROM drives. Alternatively, the disk rotation speed can stay constant; in this case, the density of bits decreases from inner tracks to outer tracks to keep the data rate constant. This method is used in hard disks and is known as constant angular velocity (CAV).

- The number of sectors per track has been increasing as disk technology improves, and the outer zone of a disk usually has several hundred sectors per track. Similarly, the number of cylinders per disk has been increasing; large disks have tens of thousands of cylinders.



## Disk Scheduling

- The **seek time** is the time for the disk arm to move the heads to the cylinder containing the desired sector.
- The **rotational latency** is the additional time for the disk to rotate the desired sector to the disk head.
- The **disk bandwidth** is the total number of bytes transferred, divided by the total time between the first request for service and the completion of the last transfer.
- We can improve both the access time and the bandwidth by managing the order in which disk I/O requests are serviced.
- For a multiprogramming system with many processes, the disk queue may often have several pending requests. Thus, when one request is completed, the operating system chooses which pending request to service next. The operating system makes this choice by any one of several disk-schedulingalgorithms.

### FCFS Scheduling

- It is the simplest of the scheduling algorithms. Consider, for example, a disk queue with requests for I/O to blocks on cylinder in that order.

98, 183, 37, 122, 14, 124, 65, 67

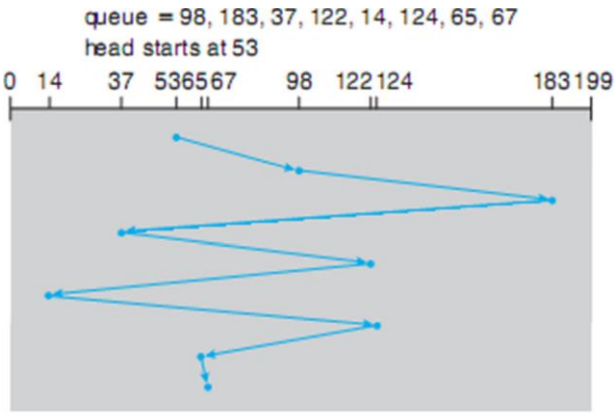



**Figure 10.4** FCFS disk scheduling.

- If the disk head is initially at cylinder 53, it will first move from 53 to 98, then to 183, 37, 122, 14, 124, 65, and finally to 67, for a total head movement of 640 cylinders.

- The wild swing from 122 to 14 and then back to 124 illustrates the problem with this schedule. If the requests for cylinders 37 and 14 could be serviced together, before or after the requests for 122 and 124, the total head movement could be decreased substantially, and performance could be thereby improved

## SSTF Scheduling

- It seems reasonable to service all the requests close to the current head position before moving the head far away to service other requests. This assumption is the basis for the shortest-seek-time-first (SSTF) algorithm. The SSTF algorithm selects the request with the least seek time from the current head position. In other words, SSTF chooses the pending request closest to the current head position.

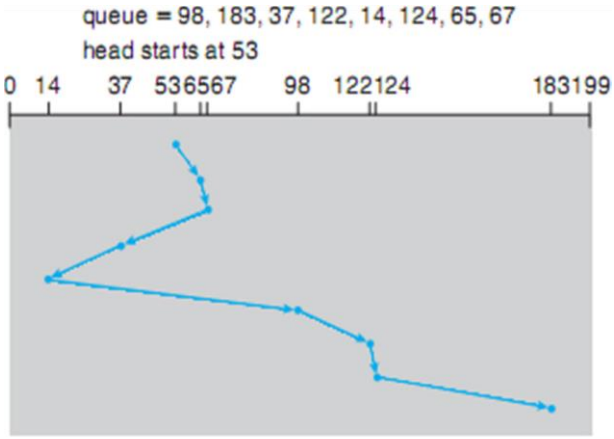



**Figure 10.5** SSTF disk scheduling.

- For our example request queue, the closest request to the initial head position (53) is at cylinder 65. Once we are at cylinder 65, the next closest request is at cylinder 67. From there, the request at cylinder 37 is closer than the one at 98, so 37 is served next. Continuing, we service the request at cylinder 14, thn 98, 122, 124, and finally 183 (Figure 10.5). This scheduling method results in a total head movement of only 236 cylinders—little more than one-third of the distance needed for FCFS scheduling of this request queue. Clearly, this algorithm gives a substantial improvement in performance
- This scheduling algorithm has the disadvantages of starvation.
- Although it is not optimal ie.. In the example, we can do better by moving the head from 53 to 37, even though the latter is not closest, and then to 14, before turning around to service 65, 67, 98, 122, 124, and 183. This strategy reduces the total head movement to 208 cylinders.

## SCAN Scheduling

- In the SCAN algorithm, the disk arm starts at one end of the disk and moves toward the other end, servicing requests as it reaches each cylinder, until it gets to the other end of the disk.
- At the other end, the direction of head movement is reversed, and servicing continues. The head continuously scans back and forth across the disk.
- The SCAN algorithm is sometimes called the elevator algorithm, since the disk arm behaves just like an elevator in a building, first servicing all the requests going up and then reversing to service requests the other way.

Example: 98, 183, 37, 122, 14, 124, 65, 67

- Before applying SCAN to schedule the requests on cylinders 98, 183, 37, 122, 14, 124, 65, and 67, we need to know the direction of head movement in addition to the head's current position.
- Assuming that the disk arm is moving toward 0 and that the initial head position is again 53, the head will next service 37 and then 14.

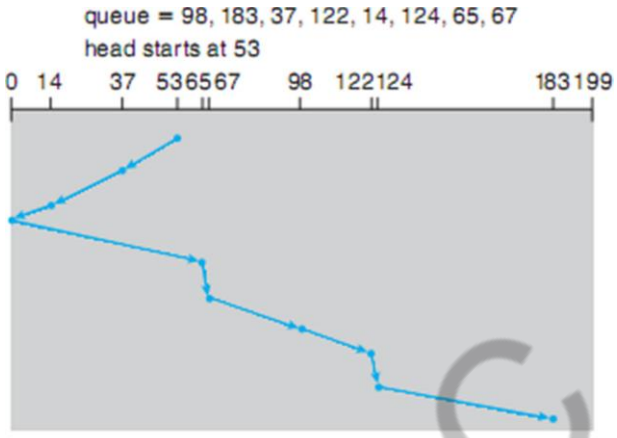



**Figure 10.6** SCAN disk scheduling.

- At cylinder 0, the arm will reverse and will move toward the other end of the disk, servicing the requests at 65, 67, 98, 122, 124, and 183.
- If a request arrives in the queue just in front of the head, it will be serviced almost immediately; a request arriving just behind the head will have to wait until the arm moves to the end of the disk, reverses direction, and comes back.

## C-SCAN Scheduling

- Circular SCAN (C-SCAN) scheduling is a variant of SCAN designed to provide a more uniform wait time. Like SCAN, C-SCAN moves the head from one end of the disk to the other, servicing requests along the way. When the head reaches the other end, however, it immediately returns to the beginning of the disk without servicing any requests on the return trip. The C-SCAN scheduling algorithm essentially treats the cylinders as a circular list that wraps around from the final cylinder to the first one

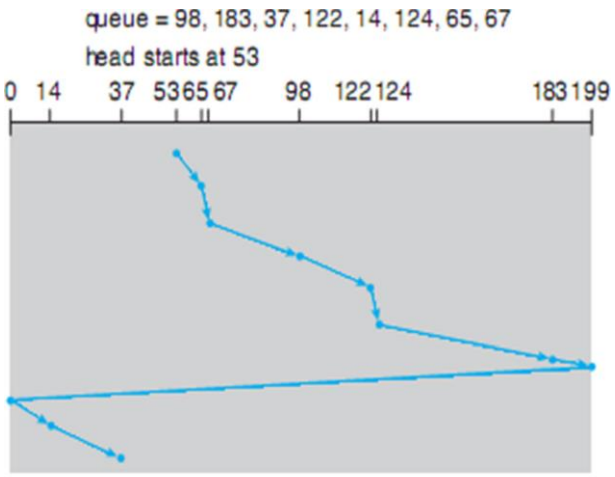



**Figure 10.7** C-SCAN disk scheduling.

## LOOK and C-LOOK Scheduling

- As we described them, both SCAN and C-SCAN move the disk arm across the full width of the disk. In practice, neither algorithm is often implemented this way.
- More commonly, the arm goes only as far as the final request in each direction. Then, it reverses direction immediately, without going all the way to the end of the disk.

- Versions of SCAN and C-SCAN that follow this pattern are called LOOK and C-LOOK scheduling, because they look for a request before continuing to move in a given direction

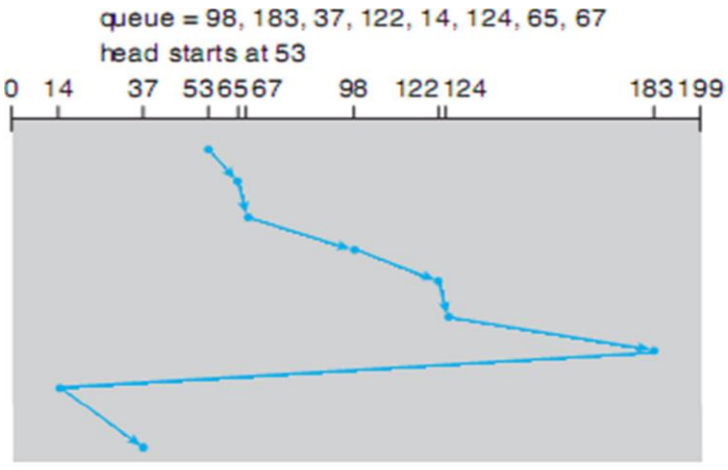



**Figure 10.8** C-LOOK disk scheduling.

**Disk Formatting**

- A new magnetic disk is a blank slate: it is just a platter of a magnetic recording material. Before a disk can store data, it must be divided into sectors that the disk controller can read and write. This process is called low-level formatting,or physical formatting.
- Low-level formatting fills the disk with a special data structure for each sector. The data structure for a sector typically consists of a header, a data area (usually 512 bytes in size), and a trailer.
- The header and trailer contain information used by the disk controller, such as a sector number and an error-correcting code (ECC).
- When the controller writes a sector of data during normal I/O, the ECC is updated with a value calculated from all the bytes in the data area. When the sector is read, the ECC is recalculated and compared with the stored value. If the stored and calculated numbers are different, the data area of the sector has become corrupted and that the disk sector may be bad.
- The ECC is an error-correcting code. If only a few bits of data have been corrupted, ECC enable the controller to identify which bits have been changed and calculate what their correct values should be.
- It then reports a recoverable soft error. The controller automatically does the ECC processing whenever a sector is read or written.
- For many hard disks, when the disk controller is instructed to low-level-format the disk, it can also be told how many bytes of data space to leave between the header and trailer of all sectors.
- It is usually possible to choose among a few sizes, such as 256, 512, and 1,024 bytes. Formatting a disk with a larger sector size means that fewer sectors can fit on each track; but it also means that fewer headers and trailers are written on each track and more space is available for user data.
- Some operating systems can handle only a sector size of 512 bytes.
- The first step is to partition the disk into one or more groups of cylinders. The operating system can treat each partition as though it were a separate disk.
- The second step is logical formatting, or creation of a file system. In this step, the operating system stores the initial file-system data structures onto the disk.
- To increase efficiency, most file systems group blocks together into larger chunks, frequently called clusters.
- Some operating systems give special programs the ability to use a disk partition as a large sequential array of logical blocks, without any file-system data structures.

## 5.2 File Concepts

➢ A file is a named collection of related information that is recorded on secondary storage.

➢ files represent programs and data.

➢ The information in a file is defined by its creator.

➢ A file has a certain defined structure which depends on its type.

- ***text file, source file, executable file***

### 5.2.1 File Attributes

➢ A file is referred to by its name.

➢ A file's attributes vary from one operating system to another but typically consist of these:

- **Name:** The symbolic file name is the only information kept in human readable form.
- **Identifier:** This unique tag, usually a number, identifies the file within the file system; it is the non-human-readable name for thefile.
- **Type:** This information is needed for systems that support different types of files.
- **Location:** This information is a pointer to a device and to the location of the file on that device.
- **Size:** The current size of the file (in bytes, words, or blocks) and possibly the maximum allowed size are included in this attribute.
- **Protection:** Access-control information determines who can do reading, writing, executing, and so on.
- **Time, date, and user identification:** This information may be kept for creation, last modification, and last use. These data can be useful for protection, security, and usage monitoring.

### 5.2.2 File Operations

➢ A file is an abstract data type.

➢ To define a file properly, it needs to consider the operations that can be performed on files.

**Creating a file:** Two steps are necessary to create a file. First, space in the file system must be found for thefile.

**Writing a file:** To write a file, we make a system call specifying both the Name of the file and the information to be written to the file. The system must keep a write pointer to the location in the file where the next write is to take place.

- **Reading a file:** To read from a file, we use a system call that specifies the name of

the file and where (in memory) the next block of the file should be put.

- **Repositioning within a file.** The directory is searched for the appropriate entry, and the current-file-position pointer is repositioned to a given value (seek).
- **Deleting a file.** To delete a file, we search the directory for the named file. Having found the associated directory entry, we release all file space, so that it can be reused by other files, and erase the directoryentry.
- **Truncating a file.** The user may want to erase the contents of a file but keep its attributes.

➢ Other common operations include

- Appending, renaming and copy
- Several pieces of information are associated with an open file.
- File pointer.
- File-open count.
- Disk location of the file.
- Access rights

### 5.2.3 File Types

➢ The operating system should recognize and support file types. If an operating system recognizes the type of a file, it can then operate on the file in reasonable ways.

➢ The system uses the extension to indicate the type of the file and the type of operations that can be done on that file.

| file type | usual extension | function |
|---|---|---|
| executable | exe, com, bin or none | ready-to-run machine-language program |
| object | obj, o | compiled, machine language, not linked |
| source code | c, cc, java, pas, asm, a | source code in various languages |
| batch | bat, sh | commands to the command interpreter |
| text | txt, doc | textual data, documents |
| word processor | wp, tex, rtf, doc | various word-processor formats |
| library | lib, a, so, dll | libraries of routines for programmers |
| print or view | ps, pdf, jpg | ASCII or binary file in a format for printing or viewing |
| archive | arc, zip, tar | related files grouped into one file, sometimes compressed, for archiving or storage |
| multimedia | mpeg, mov, rm, mp3, avi | binary file containing audio or A/V information |

### 5.2.4 File Structure

- File types can be used to indicate the internal structure of the file.
- The operating requires that an executable file have a specific structure so that it can de system termine where in memory to load the file and what the location of the first instruction is.
- None - sequence of words, bytes
- Simple record structure
  - Lines
  - Fixed length
  - Variable length
- Complex Structures
  - Formatted document
  - Relocatable load file
- Can simulate last two with first method by inserting appropriate control characters

**Internal File Structure**

- Internally, locating an offset within a file can be complicated for the operating system.
- Disk systems typically have a well-defined block size determined by the size of a sector.
- All disk I/0 is performed in units of one block (physical record), and all blocks are the same size.
- It is unlikely that the physical record size will exactly match the length of the desired logical record. Logical records may even vary in length. Packing a number of logical records into physical blocks is a common solution to this problem.

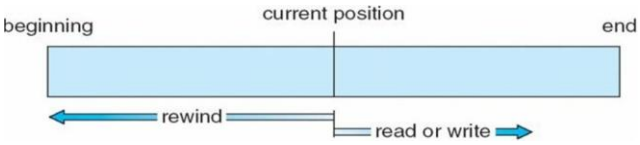
## 5.3 Access Methods

- Files store information.
- When it is used, this information must be accessed and read into computer memory.
- The information in the file can be accessed in several ways.
- Some systems provide only one access method for files while others provide many access methods.
- A major design problem is choosing one among them.

### 1. Sequential Access

- Information in the file is processed in order, one record after the other.

- Sequential Access file operations
  - **read_next()** - Read next portion of file and automatically advances a file pointer.
  - **write_next()** – Appends to the end of the file and advances to the end of the newly written material.
  - **Reset** – Back to the beginning of file
  - no read after last write




## 2. Direct Access (Relative Accesses)

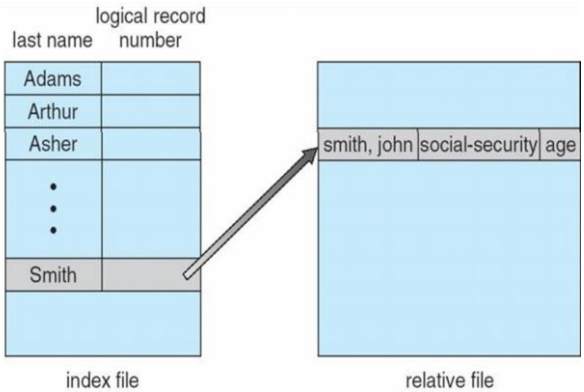
- A file is made up of fixed length logical records, that allow programs to read and write records rapidly in no particular order.
- For direct access disk is viewed as numbered sequence of blocks or record.

- Direct access file operations
  - The file operations were modified to include the block number as a parameter.
  - Read(*n)*
  - Write(*n)*
  - Position_file(n)

*n* = relative block number

## 3. Other Access Methods

- Can be built on top of basemethods
- General involve creation of an index for the file
- Keep index in memory for fast determination of location of data to be operated on (consider UPC code plus record of data about thatitem)
- If too large, index (in memory) of the index (on disk)
- To find a record, we first search the index and then use pointer to access the file directly and to find desired record.




Example of index and relative files.

## 5.4 Protection

- When information is stored in a computer system, it must be keep safe from physical damage (the issue of reliability) and improper access (the issue of protection).
- Reliability is generally provided by duplicate copies of files.
- Protection can be provided in many ways.
  - Types of Access
  - Access Control
- Other Protection Approaches

### 5.4.1 Types of Access

- Access is permitted or denied depending on several factors, one of which is the type of access requested.
- Several different types of operations may becontrolled:
  - **Read:** Read from the file.
  - **Write:** Write or rewrite thefile.
  - **Execute:** Load the file into memory and execute it.
  - **Append:** Write new information at the end of the file.
  - **Delete:** Delete the file and free its space for possible reuse.
  - **List**: List the name and attributes of the file.

### 5.4.2 Access Control

- The most common approach to the protection problem is to make access dependent on the identity of the user.
- Different users may need different types of access to a file or directory.
- To implement dependent access is to associate with each file and directory an access control list (ACL)specifying user names and the types of access allowed for each user.
- Mode of access: read, write, execute
- Many systems recognize three classifications of users in connection with each file:
  - **Owner:** The user who created the file is the owner.
  - **Group**: A set of users who are sharing the file and need similar access is a group, or work group.
  - **Universe:** All other users in the system constitute the universe.

## 5.5 File-System Implementation

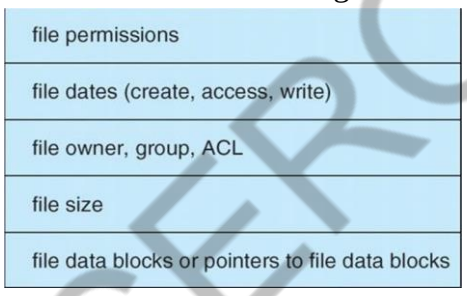
- Several on-disk and in-memory structures are used to implement a file system.
- The file system may contain information about how to boot an operating system

stored there, the total number of blocks, the number and location of free blocks, the directory structure, and individual files.

- Boot control block contains info needed by system to boot OS from that volume
  - Needed if volume contains OS, usually first block of volume
- Volume control block (superblock, master file table) contains volume details

Total number of blocks, number of free blocks, blocks size, free blocks pointers or array

- Directory structure organizes the files
  - Names and inode numbers, master file table
- Per-file File Control Block (FCB) contains many details about the file
  - inode number, permissions, size, dates
  - NFTS stores into in master file table using relational DB structures




A typical file-control block.

- The in-memory information is used for both file-system management and performance improvement via caching.
- The data are loaded at mount time, updated during file-system operations, and discarded at dismount.
- Several types of structures may be included.
  - An in-memory mount table contains information about each mounted volume.
  - An in-memory directory-structure cache holds the directory information of recently accessed directories.
  - The system wide open file table contains a copy of the FCB of each open file, as well as other information.
  - The per process open file table contains a pointer to the appropriate entry in the system-wide open-file table, as well as other information.
  - Buffers hold file-system blocks when they are being read from disk or written to disk.

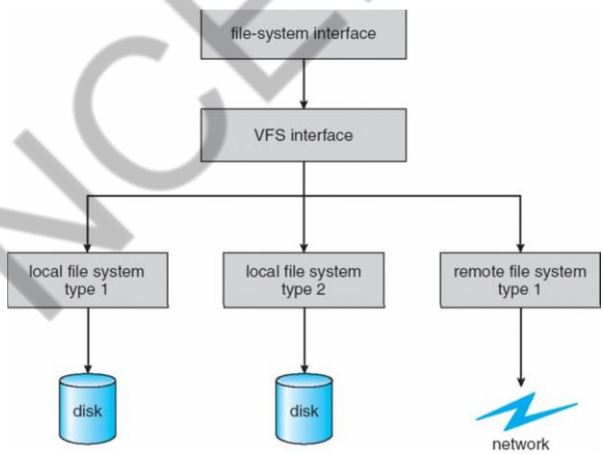
## Partitions and Mounting

- Partition can be a volume containing a file system ("cooked") or raw – just a sequence of blocks with no file system
- Boot block can point to boot volume or boot loader set of blocks that contain enough code to know how to load the kernel from the file system
  - Or a boot management program for multi-os booting

- Root partition contains the OS, other partitions can hold other Oses, other file systems, or be raw
  - Mounted at boot time
  - Other partitions can mount automatically or manually
- At mount time, file system consistency checked
  - Is all metadata correct?
    - If not, fix it, try again
    - If yes, add to mount table, allow access

## Virtual File Systems

- Virtual File Systems (VFS) on Unix provide an object-oriented way of implementing file systems
- VFS allows the same system call interface (the API) to be used for different types of file systems
  - Separates file-system generic operations from implementation details
  - Implementation can be one of many file systems types, or network file system
    - Implements **vnodes** which hold inodes or network file details
  - Then dispatches operation to appropriate file system implementation routines




Schematic view of a virtual file system.

## 5.6 Directory Implementation

➢ The selection of directory-allocation and directory-management algorithms significantly affects the efficiency, performance, and reliability of the file system

### Linear List

➢ The simplest method of implementing a directory is to use a linear list of file names with pointers to the data blocks.
➢ This method is simple to program but time-consuming toexecute.
➢ The real disadvantage of a linear list of directory entries is that finding a file requires a linear search.

### Hash Table

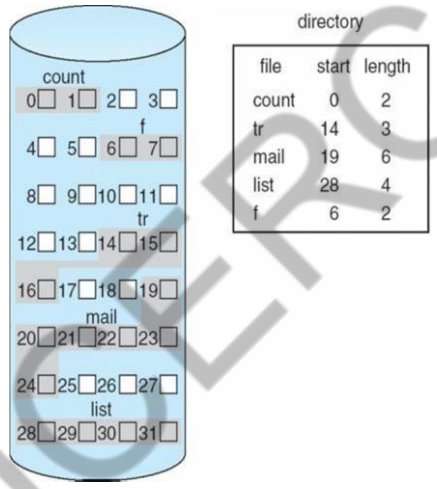
➢ With this method, a linear list stores the directory entries, but a hash data structure is also used. The hash table takes a value computed from the file name and returns a pointer to the file name in the linear list.
➢ Therefore, it can greatly decrease the directory searchtime.
➢ Insertion and deletion are also fairly straightforward, although some provision must be made for collisions-situations in which two file names hash to the same location.
➢ The major difficulties with a hash table are its generally fixed size and the dependence of the hash function on that size.
➢ a chained-overflow hash table can be used.
➢ Each hash entry can be a linked list instead of an individual value, and we can resolve collisions by adding the new entry to the linked list.

## 5.7 Allocation Methods

- An allocation method refers to how disk blocks are allocated for files.

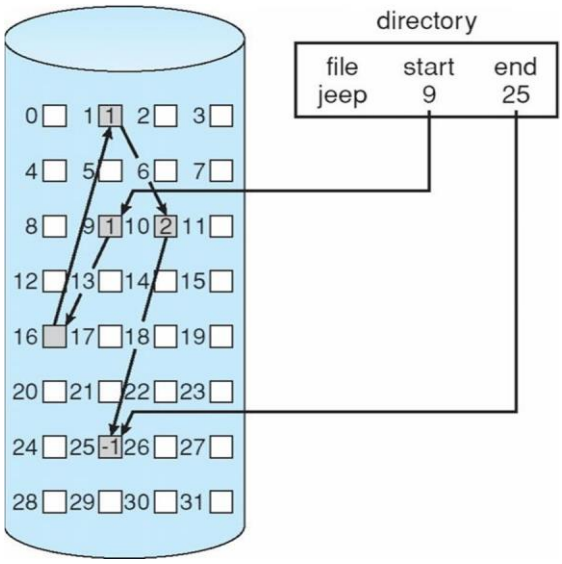
### 5.7.1 Contiguous Allocation

- requires that each file occupy a set of contiguous blocks on disk.
- Disk addresses define a linear ordering on the disk.
- Best performance in most cases
- Simple – only starting location (block number) and length (number of blocks) are required
- Problems include finding space for file, knowing file size, external fragmentation, need for **compaction off-line (downtime) or on-line**




### 5.7.2 Linked Allocation

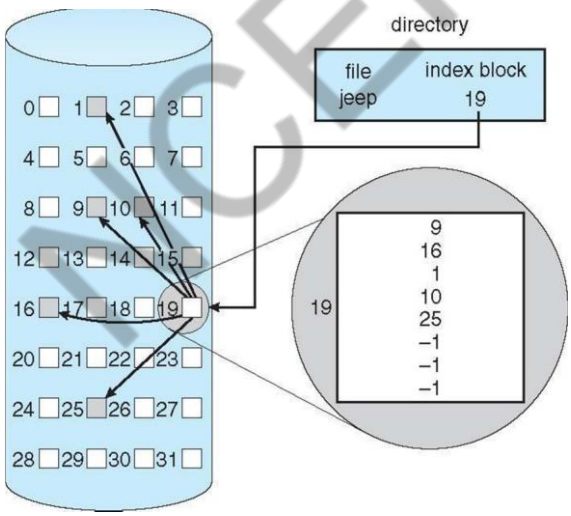
- With linked allocation, each file is a linked list of disk blocks; the disk blocks may be scattered anywhere on the disk.
- The directory points to the first and last blocks of the file.
- File ends at nil pointer
- No external fragmentation
- Each block contains pointer to next block
- No compaction, external fragmentation
- Free space management system called when new block needed
- Improve efficiency by clustering blocks into groups but increases internal fragmentation
- Reliability can be a problem
- Locating a block can take many I/Os and disk seeks




Linked allocation

### 5.7.3 Indexed allocation

- Linked allocation solves the external-fragmentation and size-declaration problems of contiguous allocation.
- Indexed allocation solves this problem by bringing all the pointers together into one location: the index block.




- Each file has its own index block, which is an array of disk-block addresses. The ith entry in the index block points to the $i^{th}$ block of the file.
- The directory contains the address of the index block. To find and read the $i^{th}$ block, we use the pointer in the $i^{th}$ index-block entry.
  - When the file is created, all pointers in the index block are set to null. When the $i^{th}$ block is first written, a block is obtained from the free-space manager, and its address is put in the $i^{th}$ index-block entry.
  - Indexed allocation supports direct access, without suffering from external fragmentation, because any free block on the disk can satisfy a request for more space. Indexed allocation does suffer from wasted space, however.

# SYSTEM PROTECTION

## GOALS OF PROTECTION

- Protection mechanisms control access to a system by limiting the types of file access permitted to users. In addition, protection must ensure that only processes that have gained proper authorization from the operating system can operate on memory segments, the CPU, and other resources.
- Protection is provided by a mechanism that controls the access of programs, processes, or users to the resources defined by a computer system. This mechanism must provide a means for specifying the controls to be imposed, together with a means of enforcing them.
- Security ensures the authentication of system users to protect the integrity of the information stored in the system (both data and code), as well as the physical resources of the computer system. The security system prevents unauthorized access, malicious destruction or alteration of data, and accidental introduction of inconsistency.
- Untrustworthy users might safely share a common logical name space, such as a directory of files, or share a common physical name space, such as memory.
- Protection is necessary to
    1. Prevent the mischievous, intentional violation of an access restriction by user.
    2. To ensure that each program component active in a system uses system resources only in ways consistent with stated policies.
    3. A protection-oriented system provides means to distinguish between authorized and unauthorized usage.
    4. The role of protection in a computer system is to provide a mechanism for the enforcement of the policies governing resource use. A protection system must have the flexibility to enforce a variety of policies.
- Note that mechanisms are distinct from policies. Mechanisms determine how something will be done; policies decide what will be done. The separation of policy and mechanism is important for flexibility. Policies are likely to change from place to place or time to time. In the worst case, every change in policy would require a change in the underlying mechanism. Using general mechanisms enables us to avoid such a situation.

## 6.2 Principles of Protection

- A key, time-tested guiding principle for protection is the principle of least privilege. It dictates that programs, users, and even systems be given just enough privileges to perform their tasks.
- Consider the analogy of a security guard with a passkey. If this key allows the guard into just the public areas that she guards, then misuse of the key will result in minimal damage. If, however, the passkey allows access to all areas, then damage from its being lost, stolen, misused, copied, or otherwise compromised will be much greater.
- The overflow of a buffer in a system daemon might cause the daemon process to fail but should not allow the execution of code from the daemon process's stack that would enable a remote user to gain maximum privileges and access to the entire system
- The creation of audit trails for all privileged function access. The audit trail allows the programmer, system administrator, or law-enforcement officer to trace all protection and security activities on thesystem.
- Managing users with the principle of least privilege entails creating a separate account for each user, with just the privileges that the user needs. An operator who needs to mount tapes and back up files on the system has access to just those commands and files needed to accomplish the job. Some systems

  implement role-based access control (RBAC) to provide this functionality.

## 6.3 Domain of Protection

- A computer system is a collection of processes and objects. By objects, we mean both hardware objects (such as the CPU, memory segments, printers, disks, and tape drives) and software objects (such as files, programs, and semaphores). Each object has a unique name that differentiates it from all other objects in the system, and each can be accessed only through well-defined and meaningful operations. Objects are essentially abstract data types.
- The operations that are possible may depend on the object.
- A process should be allowed to access only those resources for which it has authorization.
- At any time, a process should be able to access only those resources that it currently requires to complete its task. This second requirement, commonly referred to as the need-to-know principle, is useful in limiting the amount of damage a faulty process can cause in thesystem.

## 1. Domain Structure

- A protection domain, which specifies the resources that the process may access. Each domain defines a set of objects and the types of operations that may be invoked on each object.
- The ability to execute an operation on an object is an accessright. A domain is a collection of access rights, each of which is an ordered pair <object-name, rights- set>.
- For example, if domain D has the access right <file F, {read,write}>, then a process executing in domain D can both read and write file F. It cannot, however, perform any other operation on that object.
- Domains may share access rights. Ex. we have three domains: D1, D2,and D3. The access right <O4, {print}> is shared by D2 and D3, implying that a process executing in either of these two domains can print object O4.
- The association between a process and a domain may be either static, if the set of resources available to the process is fixed throughout the process's lifetime, or dynamic.
- If the association between processes and domains is fixed, and we want to adhere to the need-to-know principle, then a mechanism must be available to change the content of a domain.

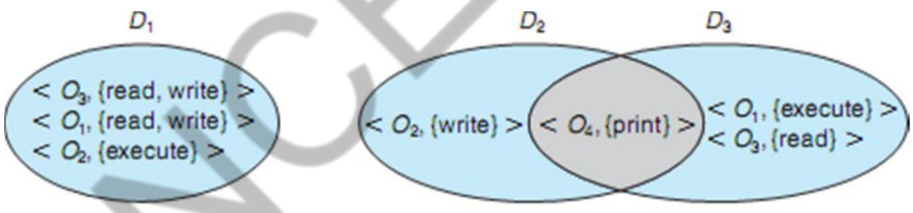



**Figure 14.1** System with three protection domains.

- If the association is dynamic, a mechanism is available to allow domain switching, enabling the process to switch from one domain to another. We may also want to allow the content of a domain to be changed. If we cannot change the content of a domain, we can provide the same effect by creating a new domain with the changed content and switching to that new domain when we want to change the domain content.
- A domain can be realized in a variety of ways:
    1. Each user may be a domain.
    2. Each process may be a domain.
    3. Each procedure may be a domain.

## 6.4 Access Matrix

- Our general model of protection can be viewed abstractly as a matrix, called an access matrix.
- The rows of the access matrix represent domains, and the columns represent objects.
- Each entry in the matrix consists of a set of access rights. Because the column defines objects explicitly, we can omit the object name from the access right. The entry access(i,j) defines the set of operations that a process executing in domain Di can invoke on object Oj.
- Consider a sample Access Matrix

| domain \ object | $F_1$ | $F_2$ | $F_3$ | printer |
|---|---|---|---|---|
| $D_1$ | read | | read | |
| $D_2$ | | | | print |
| $D_3$ | | read | execute | |
| $D_4$ | read write | | read write | |

**Figure 14.3** Access matrix.



- There are four domains and four objects—three files (F1, F2, F3) and one laser printer. A process executing in domain D1 can read files F1 and F3. A process executing in domain D4 has the same privileges as one executing in domain D1; but in addition, it can also write onto files F1 and F3.The laser printer can be accessed only by a process executing in domain D2.
- The access-matrix scheme provides us with the mechanism for specifying a variety of policies. The mechanism consists of implementing the access matrix and ensuring that the semantic properties we have outlined hold. More specifically, we must ensure that a process executing in domain Di can access only those objects specified in row i, and then only as allowed by the access- matrix entries.
- The access matrix can implement policy decisions concerning protection.
- The access matrix provides an appropriate mechanism for defining and implementing strict control for both static and dynamic association

between processes and domains.

- Processes should be able to switch from one domain to another. Switching from domain Di to domain Dj is allowed if and only if the access right switch ∈ access(i, j).
- In the following figure , a process executing in domain D2 can switch to domain D3 or to domain D4. A process in domain D4 can switch to D1,and one in domain D1 can switch to D2.

| object / domain | $F_1$ | $F_2$ | $F_3$ | laser printer | $D_1$ | $D_2$ | $D_3$ | $D_4$ |
|---|---|---|---|---|---|---|---|---|
| $D_1$ | read | | read | | | switch | | |
| $D_2$ | | | | print | | | switch | switch |
| $D_3$ | | read | execute | | | | | |
| $D_4$ | read write | | read write | | switch | | | |

**Figure 14.4** Access matrix of Figure 14.3 with domains as objects.

# Content beyond syllabus

## Familiarization of various operating systems

### Microsoft Windows




If you're looking for an operating system, you likely heard the name of Microsoft Windows. It is one of the most famous operating systems in the world. Microsoft Windows is commonly known as Windows. It is a collection of multiple proprietary graphical operating system families created and marketed by Microsoft. It allows you to store files, play games, watch videos, run software, and access the internet. Its quick navigation and user-friendly layout make it one of the top PC operating systems. To ensure security, Microsoft Windows includes antivirus and firewall.

The first version of Microsoft Windows, version 1.0, was released on November 10, 1983. Following then, more than a dozen versions of Windows were released, including the current version, Windows 10. In 2017, Windows 10 was released, and it comes in several editions, including Windows 10 Home and Pro.

#### Advantages and disadvantages of Microsoft windows

There are various advantages and disadvantages of Microsoft windows. Some of them are as follows:

**Advantages**

1. It provides high-level protection with built-in antivirus.
2. Microsoft Windows is the best operating system for beginners.
3. It is a fast-starting operating system with better application loading times.

**Disadvantages**

1. If you installed various software's; you will face many problems with this OS.
2. Windows 10 doesn't come from the windows media center, unlike the previous versions.
3. Windows 10 has received some criticism for its collecting of user data. So, privacy may be a concern.
4. It is very sensitive to malware and viruses.

# MacOS




MacOS is a series of the graphical operating system that is developed and marketed by Apple Inc. since 2001. It is Apple's main operating system for Mac computers. The original version, known informally as the classic Mac OS, was released in 1984, and the final version was Mac OS 9, which was released in 1999. Mac OS X 10.0, the first desktop version, was launched in March 2001, followed by the first update, 10.1, later that year. The most recent version is MacOS Big Sur, which was released in November 2020. The upgrade includes a system-wide dark mode and a slew of new apps ported from iOS, including Apple News. After Microsoft Windows, macOS is the second most popular desktop operating system in the market for desktop, laptop, home computers, and web usage.

his operating system was designed to operate on Apple's Mac computers. It comes with various pre-installed apps. It also permits the user to download software from the Mac AppStore. The Dark mode is a prominent feature of this operating system. It reduces overall light and brightness, making it more comfortable for the eyes. Dynamic desktop is another tool that helps in a similar way. For security, macOS requires apps to ask for permission before utilizing the camera, microphone, geolocation, or contacts. It also includes an ad-blocker for Safari. In any case, it is extremely secure. So, if you're looking for the greatest and fastest operating system, macOS is the ideal option for laptops (MacBooks) and PCs (iMac).

## Advantages and Disadvantages of MacOS

There are various advantages and disadvantages of MacOS. Some of them are as follows:

**Advantages**

1. It provides a simple and easy user-friendly interface.
2. It is the fastest operating system for the laptop and PC.
3. It comes with various pre-installed applications.
4. It provides regular security updates for the OS.
5. It also has features that make transitioning between workspaces and multitasking easier.

**Disadvantages**

1. It is only available for the MacBook and iMac.
2. MacOS devices are more costly than Windows devices.
3. There are more games and applications for Windows than for macOS.

---

# Ubuntu




Ubuntu is another best operating system for laptops and PCs. It is a free and open-source operating system that contains a firewall and virus protection software. The operating system makes it easier to use by providing completely translated versions in 50 languages. Ubuntu is regarded as one of the fastest free operating systems available. The desktop interface is simple and well-organized. It also includes various pre-installed applications, including an office suite, browsers, and media apps. More apps and games may be found in the Ubuntu Software Centre. The most recent LTS version provides five years of free security and maintenance updates. Among Linux distributions, it is regarded as the greatest operating system for laptops or computers, particularly for developers.

## Advantages and Disadvantages of Ubuntu

There are various advantages and disadvantages of Ubuntu. Some of them are as follows:

**Advantages**

1. Its LTS version provides five years of free security and maintenance updates.
2. It is used by developers.
3. It is the fastest operating system for laptops and computers.
4. It easily resolves the problem due to its big community.
5. It is free and has some basic pre-installed applications.

**Disadvantages**

1. You must find alternatives for several popular software, such as Adobe or Microsoft products because they do not provide support.

---

# Linux Fedora




It is a Linux-based operating system that competes with Ubuntu's open-source features. It is a dependable, user-friendly operating system that may run on any laptop or desktop. It is a powerful operating system that programmers widely use. It's yet another Linux distribution that's available for free. Since 2003, the Fedora Project has been working on it. Many Linux-based operating systems have a reputation for being fast. Fedora is also among the greatest operating systems for laptops and desktops.

It comes with several pre-installed open-source software. You may also use it to install third-party software. The user interface is specifically designed to eliminate distractions and help in concentration. It protects users by keeping track of all system activity. It also comes with a firewall by default, and users may quickly change the firewall settings.

## Advantages and Disadvantages of Linux Fedora

There are various advantages and disadvantages of Linux Fedora. Some of them are as follows:

**Advantages**

1. It is very lightweight and quick to access.

2. It has a shorter life cycle and is more capable of integrating new technology.
3. It comes with a lot of applications pre-installed, but its main concentration is on free software.

**Disadvantages**

1. There may be some software support concerns due to its smaller community.
2. It is not particularly beginner-friendly.

---

# Linux Mint



Linux Mint is a community-driven Linux distribution based on Ubuntu that comes with several free and open-source software. It comes with several free pre-installed applications. It also includes full media support out of the box. It is incredibly smooth, classy, and simple to use.

The most recent release of Linux Mint is Linux Mint 20, which's available in three editions. The Cinnamon version is modern and has several new features. The second is MATE. It is more stable at high speeds. MATE is a Linux distribution that is regarded as one of the quickest. Finally, Xfce is more lightweight and stable than MATE. Their most popular edition is cinnamon.

## Advantages and Disadvantages of Linux Mint

There are various advantages and disadvantages of Linux Mint. Some of them are as follows:

**Advantages**

1. It is safe and dependable because of the careful approach to software updates.
2. There are various desktop environments available.
3. It provides full multimedia support.

**Disadvantages**

1. As earlier said, it takes a careful approach to software upgrades, which can be a problem if you wish to use newer applications.
2. There is no device manager.

---

# Elementary OS

Elementary OS is a Linux operating system based on Ubuntu LTS that is known for its attractive user interface. The operating system also has a similar appearance to macOS, making it a viable alternative. Many users consider it the finest operating system for laptops because of its stability and performance.

Aside from that, it has excellent security and privacy features. If an app asks your location, you will be notified. The basic operating system also cleans up temporary files to save space. It includes a basic set of applications that you will need, such as a browser, media app, calendar, and others. More apps can be downloaded through the AppCenter. The most recent elementary operating system is 5.1 Hera, which has a screen greeter that is very useful in guiding new users.



## Advantages and Disadvantages of Elementary OS

There are various advantages and disadvantages of Elementary OS. Some of them are as follows:

### Advantages

1. It is built on Ubuntu LTS.
2. It provides customization choices with Elementary tweaks.
3. It provides MacOS like feel, so that it may be a good choice.

### Disadvantages

1. New releases and updates take a long time to arrive.
2. Slow updates may cause problems with new programs.

# Solaris


Solaris is a proprietary operating system based on UNIX. Its design emphasizes simplicity. It allows you to update the complete cloud installation with a single command for ease of maintenance. Solaris may be the ideal operating system for your PC if you are seeking something cloud-friendly. Aside from that, this operating system is well-known for its scalability.

Solaris OS is also very secure. The User and Process Rights Management reduces hacking threats by requiring users and applications to have the minimum capabilities required to complete their duties. It includes a built-in firewall for network security. The most recent release Solaris 11.4, includes the System Web Interface. This program allows you to track and view data about your current and historical system behavior.



## Advantages and Disadvantages of Solaris

There are various advantages and disadvantages of Solaris. Some of them are as follows:

### Advantages

1. It provides backup and restores utilities.
2. It provides great virus protection.
3. It supports the ZFS file system, which protects data and performs well with big amounts of data.

### Disadvantages

1. When compared to Linux Operating Systems, the hardware support is lacking.
2. It doesn't have community support.

# Solus


Solus is a Linux-Kernel-based operating system with a design aimed at providing a better home computing experience. It comes with pre-installed critical apps. The operating system is available in several editions, each with a different desktop interface. It is a choice of the homegrown Budgie desktop environment, MATE or KDE Plasma, GNOME as the desktop environment.

It allows the users to manage notifications, media devices, and other features. GNOME is easy to use and has a high level of accessibility. MATE is a more traditional desktop that caters to advanced users. Finally, Plasma is intended for users who work on or modify various aspects. Some features may be customized, like themes, clocks, and others. It provides various options for PCs and laptops.

## Advantages and Disadvantages of Solus

There are various advantages and disadvantages of Solus. Some of them are as follows:

### Advantages

1. Users get regular upgrades and don't have to worry about their operating system reaching its end-of-life.
2. The operating system is intended to be simple to use for both beginner and advanced users.
3. Users can pick and choose which updates they want to install.

### Disadvantages

1. Its software development process is slower.

# Chrome OS


Chrome OS is known for being one of the quickest operating systems available. This Chromebook OS is also very safe, reliable, and simple to use. Sandboxing is one of its security features. It means that distinct software is maintained separately so that if one element becomes infected, the rest of the system remains safe and secure. It also has an antivirus program built-in.

This Linux-kernel-based operating system's primary user interface is Google Chrome. Chrome OS only supports web capabilities and does not run system checks, making it one of the fastest operating systems available. It is compatible with both Android and Linux applications.



## Advantages and Disadvantages of Chrome OS

There are various advantages and disadvantages of Chrome OS. Some of them are as follows:

### Advantages

1. It supports Android Apps.
2. Its devices are cheaper than macOS devices.
3. It is lightweight and fast.
4. It is the fastest operating system for laptops and computers.
5. You don't have to worry about it because the operating system automatically backs up your data to the cloud.

### Disadvantages

1. Most of the applications are available online. As a result, most jobs rely on the internet.
2. It is also extremely limited, making it unsuitable for advanced users.

3. The collecting of data by Chrome OS can be a privacy concern.

---

# CentOS


CentOS is yet another community-driven free and open-source software platform that enables powerful platform management. It is ideal for developers looking for an operating system that simply assists them in their coding chores. That isn't to say it has nothing to offer individuals who only want to use it for daily tasks.

## Feature of CentOS

There are various features of CentOS. Some of them are as follows:

1. It offers the most powerful security capabilities available, such as process and user rights control, allowing you to protect mission-critical data.
2. Advanced networking, compatibility, and security features are still lacking in many operating systems today.
3. It enables seamless interchange by addressing hundreds of hardware and software issues.

---