DIGITAL NOTES
ON
OPERATING SYSTEMS

B.TECH III YEAR - I SEM
(2019-20)

DEPARTMENT OF INFORMATION TECHNOLOGY

MALLA REDDY COLLEGE OF ENGINEERING & TECHNOLOGY
(Autonomous Institution – UGC, Govt. of India)
(Affiliated to JNTUH, Hyderabad, Approved by AICTE - Accredited by NBA & NAAC – 'A' Grade - ISO 9001:2015 Certified)
Maisammaguda, Dhulapally (Post Via. Hakimpet), Secunderabad – 500100, Telangana State, INDIA.

SYLLABUS

III Year B.Tech IT –I Sem

L	T /P/D	C
3	- / - / -	3

(R17A0513)OPERATING SYSTEMS

Objectives:

- To understand main components of OS and their working
- To study the operations performed by OS as a resource manager
- To analyze the different scheduling policies of OS
- To be aware of the different memory management techniques
- To understand process concurrency and synchronization
- To comprehend the concepts of input/ output, storage and file management

UNIT - I:

Operating System Introduction: Operating Systems Objectives and functions, Computer System Architecture, OS Structure, OS Operations, Evolution of Operating Systems - Simple Batch, Multi programmed, time shared, Personal Computer, Parallel, Distributed Systems, Real-Time Systems, Special - Purpose Systems, Operating System services, user OS Interface, System Calls, Types of System Calls, System Programs, Operating System Design and Implementation, OS Structure, Virtual machines.

UNIT - II:

Process and CPU Scheduling - Process concepts - The Process, Process State, Process Control Block, Threads, Process Scheduling - Scheduling Queues, Schedulers, Context Switch, Preemptive Scheduling, Dispatcher, Scheduling Criteria, Scheduling algorithms, Multiple-Processor Scheduling, Real-Time Scheduling, Thread scheduling, Case studies: Linux, Windows.

Process Coordination - Process Synchronization, The Critical section Problem, Peterson's solution, Synchronization Hardware, Semaphores, and Classic Problems of Synchronization, Monitors, Case Studies: Linux, Windows.

UNIT - III:

Memory Management and Virtual Memory - Logical & physical Address Space, Swapping, Contiguous Allocation, Paging, Structure of Page Table. Segmentation, Segmentation with Paging, Virtual Memory, Demand Paging, Performance of Demanding Paging, Page Replacement - Page Replacement Algorithms, Allocation of Frames, Thrashing.

UNIT - IV:

File System Interface - The Concept of a File, Access methods, Directory Structure, File System Mounting, File Sharing, Protection, File System Implementation - File System Structure, File System Implementation, Allocation methods, Free-space Management, Directory Implementation, Efficiency and Performance.

Mass Storage Structure - Overview of Mass Storage Structure, Disk Structure, Disk Attachment, Disk Scheduling, Disk Management, Swap space Management.

UNIT - V:

Deadlocks - System Model, Deadlock Characterization, Methods for Handling Deadlocks, Deadlock Prevention, Deadlock Avoidance, Deadlock Detection, Recovery from Deadlock.

Protection - System Protection, Goals of Protection, Principles of Protection, Domain of Protection, Access Matrix, Implementation of Access Matrix, Access Control, Revocation of Access Rights, Capability-Based Systems, Language-Based Protection.

TEXT BOOKS:

1. Operating System Principles, Abraham Silberchatz, Peter B. Galvin, Greg Gagne 8th Edition, Wiley Student Edition.

2. Operating systems - Internals and Design Principles, W. Stallings, 6th Edition, Pearson.

REFERENCES BOOKS:

1. Modern Operating Systems, Andrew S Tanenbaum 3rd Edition PHI.
2. Operating Systems A concept - based Approach, 2nd Edition, D. M. Dhamdhere, TMH.
3. Principles of Operating Systems, B. L. Stuart, Cengage learning, India Edition.
4. Operating Systems, A. S. Godbole, 2nd Edition, TMH
5. Operating Systems, S, Haldar and A. A. Arvind, Pearson Education.
6. Operating Systems, R. Elmasri, A. G. Carrick and D. Levine, Mc Graw Hill.

Outcome:

- Apply optimization techniques for the improvement of system performance.
- Ability to understand the synchronous and asynchronous communication mechanisms in their respective OS.
- Learn about minimization and maximization of relevant performance criteria.
- Ability to compare the features of different OS

INDEX

UNIT I:

Operating System Introduction: Operating Systems Objectives and functions, Computer System Architecture, OS Structure, OS Operations, Evolution of Operating Systems - Simple Batch, Multi programmed, time shared, Personal Computer, Parallel, Distributed Systems, Real-Time Systems, Special - Purpose Systems, Operating System services, user OS Interface, System Calls, Types of System Calls, System Programs, Operating System Design and Implementation, OS Structure, Virtual machines.

OPERATING SYSTEMS OBJECTIVES AND FUNCTIONS

An OS is a program that controls the execution of application programs and acts as an interface between applications and the computer hardware. It can be thought of as having three objectives :

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

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

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

A computer is a set of resources for the movement, storage, and processing of data and for the control of these functions

The OS functions in the same way as ordinary computer software; that is, it is a program or suite of programs executed by the processor.

The OS frequently relinquishes control and must depend on the processor to allow it to regain control.

COMPUTER SYSTEM ARCHITECTURE

OS typically provides services in the following areas:

Program development: The OS provides a variety of facilities and services, such as editors and debuggers, to assist the programmer in creating programs.

Typically, these services are in the form of utility programs that, while not strictly part of the core of the OS, are supplied with the OS and are referred to as application program development tools.

Program execution: A number of steps need to be performed to execute a program. Instructions and data must be loaded into main memory, I/O devices

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

Single-Processor Systems:

On a singleprocessor system, there is one main CPUcapable of executing a general-purpose instruction set, including instructions from user processes. Almost all singleprocessor systems have other special-purpose processors as well. They may come in the form of device-specific processors, such as disk, keyboard, and graphics controllers; or, on mainframes, they may come in the form of more general-purpose processors, such as I/O processors that move data rapidly among the components of the system.

Multiprocessor Systems: multiprocessor systems (also known as parallelsystems or multicore systems) have two or more processors in close communication, sharing the computer bus and sometimes the clock, memory, and peripheral devices. Multiprocessor systems have three main

advantages:

1. **Increased throughput**. By increasing the number of processors,we expect to get more work done in less time.
2. **Economy of scale**. Multiprocessor systems can cost less than equivalent multiple single-processor systems, because they can share peripherals, mass storage, and power supplies
3. **Increased reliability**. If functions can be distributed properly among several processors, then the failure of one processor will not halt the system, only slow it down

multiple-processor systems in use today are of two types. Some systems use asymmetric multiprocessing, in which each processor is assigned a specific task.Aboss processor controls the system; the other processors either look to the boss for instruction or have predefined tasks

symmetric multiprocessing (SMP), in which each processor performs all tasks within the operating system. SMP means that all processors are peers; no boss–worker relationship exists between processors

Multiprocessing adds CPUs to increase computing power. If the CPU has an integrated memory controller, then adding CPUs can also increase the amount. multiprocessing can cause a system to change its memory access model from uniform memory access (UMA) to non-uniform memory access (NUMA). Multiprocessor systems are termed multicore that has CPU design is to include multiple computing cores on a single chip

Clustered Systems: clustered system, which gathers together multiple CPUs. Clustered systems differ from the multiprocessor systems in that they are composed of two or more individual systems—or nodes—joined together. Such systems are considered loosely coupled. .

OS STRUCTURE

An operating system provides the environment within which programs are executed.

A **single program** cannot, in general, keep either the CPU or the I/O devices busy at all times. Single users frequently have multiple programs running.

Multiprogramming increases CPUutilization by organizing jobs (code and data) so that the CPU always has one to execute

Main memory is too small to accommodate all jobs, the jobs are kept initially on the disk in the job pool. This pool consists of all processes residing on disk awaiting allocation of main memory.

Multiprogrammed systems provide an environment in which the various system resources (for example, CPU, memory, and peripheral devices) are utilized effectively.

In **Time sharing (or multitasking)** the CPU executes multiple jobs by switching among them, but the switches occur so frequently that the users can interact with each program while it is running. A time-shared operating system allows many users to share the computer simultaneously. Since each action or command in a time-shared system tends to be short, the response time should be short—typically less than one second.

Time sharing and multiprogramming require that several jobs be kept simultaneously in memory. If several jobs are ready to be brought into memory, and if there is not enough room for all of them, then the system must choose among them. Making this decision involves job scheduling. several jobs are ready to run at the same time, the system must choose which job will run first. Making this decision is CPUscheduling

In swapping, whereby processes are swapped in and out of main memory to the disk.A more common method for ensuring reasonable response time is virtual memory, a technique that allows the execution of a process that is not completely in memory. The main advantage of the virtual-memory scheme is that it enables users to run programs that are larger than actual physical memory. It abstracts main memory into a large, uniform array of storage, separating logical memory

OS 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.

Dual-Mode and Multimode Operation

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. At system boot time, the hardware starts in kernel mode.

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 concept of modes can be extended beyond two modes. 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.

Timer: 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.

EVOLUTION OF OPERATING SYSTEMS

It is useful to consider how operating systems have evolved over the years.

Serial Processing: With the earliest computers, from the late 1940s to the mid-1950s, the programmer interacted directly with the computer hardware; there was no OS. These computers were run from a console consisting of display lights, toggle switches, some form of input device, and a printer. Programs in machine code were loaded via the input device (e.g., a card reader). If an error halted the program, the error condition was indicated by the lights. The mode of operation could be termed serial processing, reflecting the fact that users have access to the computer in series.

Simple Batch Systems:To improve utilization, the concept of a batch OS was developed. It appears that the first batch OS (and the first OS of any kind) was developed in the mid-1950s by General Motors for use on an IBM 701.The central idea behind the simple batch-processing scheme is the use of a piece of software known as the monitor. With this type of OS, the user no longer has direct access to the processor. Instead, the user submits the job on cards or tape to a computer operator, who batches the jobs together sequentially and places the entire batch on an input device, for use by the monitor. With each job, instructions are included in a primitive form of job control language (JCL). This is a special type of programming language used to provide instructions to the monitor.

The monitor, or batch OS, is simply a computer program. It relies on the ability of the processor to fetch instructions from various portions of main memory to alternately seize and relinquish control.

Multiprogrammed Batch Systems: Suppose that there is room for the OS and two user programs. When one job needs to wait for I/O, the processor can switch to the other job, which is likely not waiting for I/O. Furthermore, we might expand memory to hold three, four, or more programs and switch among all of them. The approach is known as multiprogramming, or multitasking

Time-Sharing Systems: Just as multiprogramming allows the processor to handle multiple batch jobs at a time, multiprogramming can also be used to handle multiple interactive jobs. In this latter case, the technique is referred to as time sharing, because processor time is shared among multiple users.

PERSONAL COMPUTER (PC) SYSTEMS

A computer system dedicated to a single user is referred to as a PC. In the first PCs, the operating system was neither multiuser nor multitasking (eg. MS-DOS). The operating system concepts used in mainframes and minicomputers, today, are also used in PCs (eg. UNIX, Microsoft Windows NT, Macintosh OS).

PARALLEL SYSTEMS

Parallel systems have more than one processor. In multiprocessor systems (tightly coupled), processors are in close communication, such as they share computer bus, clock, memory or peripherals.

DISTRIBUTED SYSTEMS

Distributed systems also have more than one processor. Each processor has its local memory. Processors communicate through communication lines (eg. Telephone lines, high-speed bus, etc.). Processors are referred to as sites, nodes, computers depending on the context in which they are mentioned. Multicomputer systems are loosely coupled. Example applications are e-mail, web server, etc.

REAL-TIME SYSTEMS

Real-time systems are special purpose operating systems. They are used when there are rigid time requirements on the operation of a processor or the flow of data, and thus it is often used as a control device in a dedicated application (eg. fuel injection systems, weapon systems, industrial control systems, …). It has well defined, fixed time constraints. The processing must be done within the defined constraints, or the system fails.

Two types:
- Hard real-time systems guarantee that critical tasks complete on time.
- In Soft real-time systems, a critical real-time task gets priority over other tasks, and the task retains that priority until it completes.

OPERATING SYSTEM SERVICES

An operating system provides an environment for the execution of programs. It provides certain services to programs and to the users of those programs.
One set of operating system services provides functions that are helpful to the user.
User interface. Almost all operating systems have a user interface (UI). This interface can take several forms. One is a command-line interface (CLI), which uses text commands and a method for entering them (say, a keyboard for typing in commands in a specific format with specific options). Another is a batch interface, in which commands and directives to control those commands are entered into files, and those files are executed. Most commonly, a graphical user interface (GUI) is used.
Program execution. The system must be able to load a program into memory and to run that program I/O operations. A running program may require I/O, which may involve a file or an I/O device.
File-system manipulation. The file system is of particular interest. Obviously, programs need to read and write files and directories. They also need to create and delete them by name, search for a given file, and list file information. operating systems include permissions management to allow or deny access to files or directories based on file ownership
Communications: Communications may be implemented via shared memory, in which two or more processes read and write to a shared section of memory, or message passing, in which

packets of information in predefined formats are moved between processes by the operating system.

Error detection. The operating system needs to be detecting and correcting errors constantly. Errorsmay occur in the CPU and memory hardware (such as a memory error or a power failure), etc.

Systems with multiple users can gain efficiency by sharing the computer resources among the users.

Resource allocation. When there are multiple users or multiple jobs running at the same time, resourcesmust be allocated to each of them. The operating system manages many different types of resources.

Protection and security. The owners of information stored in a multiuser or networked computer system may want to control use of that information.

USER OS INTERFACE

There are several ways for users to interface with the operating system. Here, we discuss two fundamental approaches. One provides a command-line interface, or command interpreter, that allows users to directly enter commands to be performed by the operating system. The other allows users to interface with the operating system via a graphical user interface, or GUI

SYSTEM CALLS

System calls provide an interface to the services made available by an operating system. These calls are generally available as routines written in C and C++, although certain low-level tasks (for example, tasks where hardware must be accessed directly) may have to be written using assembly-language instructions.

TYPES OF SYSTEM CALLS

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

A running program needs to be able to halt its execution either normally (end()) or abnormally (abort()). If a system call is made to terminate the currently running program abnormally, or if the program runs into a problem

and causes an error trap, a dump of memory is sometimes taken and an error message generated. The dump is written to disk and may be examined by a debugger—a system program designed to aid the programmer in finding and correcting errors, or bugs—to determine the cause of the problem

File Management :System calls that need to be able to create() and delete() files. Either system call requires the name of the file and perhaps some of the file's attributes. Once the file is created, we need to open() it and to use it. We may also read(), write(), or reposition() (rewind or skip to the end of the file, for example). Finally, we need to close() the file, indicating that we are no longer using it.

Device Management: A system with multiple users may require us to first request() a device, to ensure exclusive use of it. After we are finished with the device, we release() it. These functions are similar to the open() and close() system calls for files. Once the device has been requested (and allocated to us), we can read(),

write(), and (possibly) reposition() the device, just as we can with files.

Information Maintenance: Many system calls exist simply for the purpose of transferring information between the user program and the operating system. time() and date(). dump() - This provision is useful for debugging. A program trace lists each system call as it is executed.

Communication: There are two common models of interprocess communication: the messagepassing model and the shared-memory model. message-passing model, the communicating processes exchange messages with one another to transfer information. process name, and this name is translated into an identifier by which the operating system can refer to the process. The get hostid() and get processid() system callsshared-memory model, processes use shared memory create() and shared memory attach()

Protection: Protection provides a mechanism for controlling access to the resources provided by a computer system.set permission() and get permission(), which manipulate the permission settings of resources such as files and disks. The allow user() and deny user() system calls specify whether particular users can—or cannot—be allowed access to certain resources.

SYSTEM PROGRAMS

System programs, also known as system utilities, provide a convenient environment for program development and execution. They can be divided into these categories: File management , Status information, File modification, Programming-language support, Program loading and execution, Communications, Background services

OPERATING SYSTEM DESIGN AND IMPLEMENTATION

Design Goals:
The requirements can, however, be divided into two basic groups: user goals and system goals.
User Goals:Users want certain obvious properties in a system. The system should be convenient to use, easy to learn and to use, reliable, safe, and fast. Of course, these specifications are not particularly useful in the system design, since there is no general agreement on how to achieve them.
System Goals:The system should be easy to design, implement, and maintain; and it should be flexible, reliable, error free, and efficient.
Mechanisms and Policies:One important principle is the separation of policy from mechanism. Mechanisms determine how to do something; policies determine what will be done. For example, the timer construct (see Section 1.5.2) is a mechanism for ensuring CPU protection, but deciding how long the timer is to be set for a particular user is a policy decision.

Implementation: Once an operating system is designed, it must be implemented.Early operating systems were written in assembly language.Now, although some operating systems are still written in assembly language, most are written in a higher-level language such as C or an even higher-level language.

OPERATING SYSTEM STRUCTURE

A system as large and complex as a modern operating system must be engineered carefully if it is to function properly and be modified easily. A common approach is to partition the task into small components, or modules, rather than have one monolithic system.
Simple Structure: OS started as small, simple, and limited systems and then grew beyond their original scope. MS- DOS is an example of such a system.
Layered Approach: layered approach, in which the operating system is broken into a number of layers (levels). The bottom layer (layer 0) is the hardware; the highest (layer N) is the user interface.
Microkernels: In the mid-1980s, researchers at Carnegie Mellon University developed an operating system called Mach that modularized the kernel using the microkernel approach. This method structures the operating system by removing all nonessential components fromthe kernel and implementing them as system and user-level programs. The result is a smaller kernel.

Modules: operating-system design involves using loadable kernel modules. Here, the kernel has a set of core components and links in additional services via modules, either at boot time or during run time. This type of design is common in modern implementations of UNIX, such as Solaris, Linux, and Mac OS X, as well as Windows

Hybrid Systems: In practice, very few operating systems adopt a single, strictly defined structure. Instead, they combine different structures, resulting in hybrid systems that address performance, security, and usability issues. For example, both Linux and Solaris are monolithic, because having the operating system in a single address space provides very efficient performance. However, they are also modular, so that new functionality can be dynamically added to the kernel.

Mac OS X : The Apple Mac OS X operating system uses a hybrid structure. It hass a layered system.

iOS: iOS is a mobile operating system designed by Apple to run its smartphone, the iPhone, as well as its tablet computer, the iPad.

Android: The Android operating system was designed by the Open Handset Alliance (led primarily by Google) and was developed for Android smartphones and tablet computers.

VIRTUAL MACHINES

An alternative approach is to recognize that with the ever-increasing number of cores on a chip, the attempt to multiprogram individual cores to support multiple applications may be a misplaced use of resources. If instead, we allow one or more cores to be dedicated to a particular process and then leave the processor alone to devote its efforts to that process, we avoid much of the overhead of task switching and scheduling decisions. The multicore OS could then act as a hypervisor that makes a high-level decision to allocate cores to applications but does little in the way of resource allocation beyond that.

UNIT - II:

Process and CPU Scheduling - Process concepts - The Process, Process State, Process Control Block, Threads, Process Scheduling - Scheduling Queues, Schedulers, Context Switch, Preemptive Scheduling, Dispatcher, Scheduling Criteria, Scheduling algorithms, Multiple-Processor Scheduling, Real-Time Scheduling, Thread scheduling, Case studies: Linux, Windows.

Process Coordination - Process Synchronization, The Critical section Problem, Peterson's solution, Synchronization Hardware, Semaphores, and Classic Problems of Synchronization, Monitors, Case Studies: Linux, Windows.

A process is the unit of work in a modern time-sharing system

THE PROCESS

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

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. A process may be in one of the following states:

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

Process Control Block

Each process is represented in the operating system by a process control block (PCB)—also called a task control block.APCB. It contains many pieces of information associated with a specific process, including these

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

- **Program counter**. The counter indicates the address of the next instruction to be executed for this process.
- **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**. This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters.
- **Memory-management information**. This information may include such items as the value of the base and limit registers and the page tables, or the segment tables, depending on the memory system used by the operating system.
- **Accounting information**. This information includes the amount of CPU and real time used, time limits, account numbers, job or process numbers, and so on.
- **I/O status information**. This information includes the list of I/O devices allocated to the process, a list of open files, and so on.

Threads

The process model discussed so far has implied that a process is a program that performs a single

thread of execution. For example, when a process is running a word-processor program, a single thread of instructions is being executed.

Process Scheduling

The objective of multiprogramming is to have some process running at all times, to maximize CPU utilization. The objective of time sharing is to switch the CPU among processes so frequently that users can interact with each program

while it is running. To meet these objectives, the process scheduler selects an available process (possibly from a set of several available processes) for program execution on the CPU. For a single-processor system, there will never be more than one running process. If there are more processes, the rest will have to wait until the CPU is free and can be rescheduled.

Scheduling Queues

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

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

- The process could issue an I/O request and then be placed in an I/O queue.
- The process could create a new child process and wait for the child's termination.
- The process could be removed forcibly from the CPU, as a result of an interrupt, and be put 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 this pool and loads them into memory for execution. The short-term scheduler, or CPU scheduler, selects from among the processes that are ready to execute and allocates the CPU to one of them.

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

An I/O-bound process is one that spends more of its time doing I/O than it spends doing computations. A CPU- bound process, in contrast, generates I/O requests infrequently, using more of its time doing computations. This medium-term scheduler is diagrammed. The key idea behind a medium-term scheduler is that sometimes it can be advantageous to remove a process from memory (and from active contention for the CPU) and thus reduce the degree of multiprogramming. Later, the process can be reintroduced into memory, and its execution can be continued where it left off. This scheme is called swapping.

Context Switch

When an interrupt occurs, the system needs to save the current context of the process running on the CPU so that

it can restore that context when its processing is done, essentially suspending the process and then resuming it. Switching the CPU to another process requires performing a state save of the current process and a state restore of a different process. This task is known as a context switch

CPU scheduling is the basis of multiprogrammed operating systems. Sche duling of this kind is a fundamental operating-system function.

CPU Scheduler

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

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.

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

Different CPU-scheduling algorithms have different properties, and the choice of a particular algorithm may favor one class of processes over another.

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

SCHEDULING ALGORITHMS

CPU scheduling deals with the problem ofdeciding which of the processes in the ready queue is to be allocated the CPU.There aremanydifferent 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.

Shortest-Job-First Scheduling

Adifferent approach toCPU scheduling is the shortest-job-first (SJF) scheduling algorithm. 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.

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.

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.

Multilevel Queue Scheduling

Another class of scheduling algorithms has been created for situations in which processes are easily classified into different groups. For example, a common division is made between foreground (interactive) processes and background (batch) processes.

A multilevel queue scheduling algorithm partitions the ready queue into several separate queues (Figure 6.6). The processes are permanently assigned to one queue, generally based on some property of the process, such as memory size, process priority, or process type. Each queue has its own scheduling algorithm.

Multilevel Feedback Queue Scheduling

The multilevel feedback queue scheduling algorithm, in contrast, allows a process to move between queues. The idea is to separate processes according to the characteristics of their CPU bursts.

THREAD SCHEDULING

We introduced threads to the process model, distinguishing between user-level and kernel-level threads.

Contention Scope

One distinction between user-level and kernel-level threads lies in how they are scheduled. On systems implementing the many-to-one and many-to-many models, the thread library schedules user-level threads to run on an available LWP. This scheme is known as process contention scope (PCS), since competition for the CPU takes place among threads belonging to the same process.

Pthread Scheduling

We provided a sample POSIX Pthread program, along with an introduction to thread creation with Pthreads. Now, we highlight the POSIX
Pthread API that allows specifying PCS or SCS during thread creation. Pthreads identifies the following contention scope values:

- PTHREAD SCOPE PROCESS schedules threads using PCS scheduling.
- PTHREAD SCOPE SYSTEM schedules threads using SCS scheduling.

On systems implementing the many-to-many model, the PTHREAD SCOPE PROCESS policy schedules user-level threads onto available LWPs. The number of LWPs is maintained by the thread library, perhaps using scheduler activations. The PTHREAD SCOPE SYSTEM scheduling policy will create and bind an LWP for each user-level thread on many-to-many systems, effectively mapping threads using the one-to-one policy.

The Pthread IPC provides two functions for getting—and setting—the contention scope policy:

- pthread attr setscope(pthread attr t *attr, int scope)
- pthread attr getscope(pthread attr t *attr, int *scope)

MULTIPLE-PROCESSOR SCHEDULING

If multiple CPUs are available, load sharing becomes possible—but scheduling problems become correspondingly more complex. Many possibilities have been tried; and as we saw with singleprocessor CPU scheduling, there is no one best solution

Processor Affinity

Consider what happens to cache memory when a process has been running on a specific processor. The data most recently accessed by the process populate the cache for the processor. As a result, successive memory accesses by the process are often satisfied in cache memory. Now consider what happens if the process migrates to another processor. The contents of cache memory must be invalidated for the first processor, and the cache for the second processor must be repopulated.
This is known as processor affinity—that is, a process has an affinity for the processor on which it is currently running

Load balancing

attempts to keep the workload evenly distributed across all processors in an SMP system. It is important to note that load balancing is typically necessary only on systems where each processor has its own private queue of eligible processes to execute.

Multicore Processors

hardware has been to place multiple processor cores on the same physical chip, resulting in a multicore processor. Each core maintains its architectural state and thus appears to the operating system to be a separate physical processor. SMP systems that use multicore processors are faster and consume less power
than systems in which each processor has its own physical chip.

Real-Time CPU Scheduling

CPU scheduling for real-time operating systems involves special issues. In general, we can distinguish between soft real-time systems and hard real-time systems. Soft real-time systems provide no guarantee as to when a critical real- time process will be scheduled. They guarantee only that the process will be given preference over noncritical processes. Hard real-time systems have stricter requirements. A task must be serviced by its deadline; service after the deadline has expired is the same as no service at all. In this section, we explore several issues related to process scheduling in both soft and hard real-time operating systems.

EXAMPLE: LINUX SCHEDULING

Process scheduling in Linux has had an interesting history. Prior to Version 2.5, the Linux kernel ran a variation of the traditional UNIX scheduling algorithm With Version 2.5 of the kernel, the scheduler was overhauled to include a scheduling algorithm—known as O(1)—that ran in constant time regardless of the number of tasks in the system. The O(1) scheduler also provided increased support for SMP systems, including processor affinity and load balancing between processors. However, in practice, although the O(1)scheduler delivered excellent performance on SMP systems, it led to poor response times for the interactive processes that are common on many desktop computer systems. During development of the 2.6 kernel, the scheduler was again revised; and in release 2.6.23 of the kernel, the Completely Fair Scheduler (CFS) became the default Linux scheduling algorithm

Scheduling in the Linux system is based on scheduling classes. Each class is assigned a specific priority. By using different scheduling classes, the kernel can accommodate different scheduling algorithms based on the needs of the system and its processes. The scheduling criteria for a Linux server, for example, may be different from those for a mobile device running Linux. To decide which task to run next, the scheduler selects the highest-priority task belonging to the highest-priority scheduling class. Standard Linux kernels implement two scheduling classes: (1) a default scheduling class using the CFS scheduling algorithm and (2) a real-time scheduling class

EXAMPLE: WINDOWS SCHEDULING

Windows schedules threads using a priority-based, preemptive scheduling algorithm. The Windows scheduler ensures that the highest-priority thread will always run. The portion of the Windows kernel that handles scheduling is called the dispatcher. A thread selected to run by the dispatcher will run until it is preempted by a higher-priority thread, until it terminates, until its time quantum ends, or until it calls a blocking system call, such as for I/O. If a higher-priority real-time thread becomes ready while a lower-priority thread is running, the lower-priority thread will be preempted. This preemption gives a real-time thread preferential access to the CPU when the thread needs such access.

The dispatcher uses a 32-level priority scheme to determine the order of thread execution. Priorities are divided into two classes. The variable class contains threads having priorities from 1 to 15, and the real-time class contains threads with priorities ranging from 16 to 31. (There is also a thread running at priority 0 that is used for memory management.) The dispatcher uses a queue for each scheduling priority and traverses the set of queues from highest to lowest until it finds a thread that is ready to run. If no ready thread is found, the dispatcher will execute a special thread called the idle thread. There is a relationship between the numeric priorities of the Windows kernel and the Windows API. The Windows API identifies the following six priority classes to which a process can belong:

- IDLE PRIORITY CLASS
- BELOW NORMAL PRIORITY CLASS
- NORMAL PRIORITY CLASS
- ABOVE NORMAL PRIORITY CLASS

• HIGH PRIORITY CLASS
• REALTIME PRIORITY CLASS

Processes are typically members of the NORMAL PRIORITY CLASS. A process belongs to this class unless the parent of the process was a member of the IDLE PRIORITY CLASS or unless another class was specified when the process was created. Additionally, the priority class of a process can be altered with the SetPriorityClass() function in theWindows API. Priorities in all classes except the REALTIME PRIORITY CLASS are variable, meaning that the priority of a thread belonging to one of these classes can change.

A thread within a given priority classes also has a relative priority. The values for relative priorities include:

• IDLE
• LOWEST
• BELOW NORMAL
• NORMAL
• ABOVE NORMAL
• HIGHEST
• TIME CRITICAL

The priority of each thread is based on both the priority class it belongs to and its relative priority within that class.

PROCESS SYNCHRONIZATION

A cooperating process is one that can affect or be affected by other processes executing in the system. Cooperating processes can either directly share a logical address space (that is, both code and data) or be allowed to share data only through files or messages.

A situation like this, where several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place, is called a race condition.

THE CRITICAL-SECTION PROBLEM

We begin our consideration of process synchronization by discussing the socalled critical-section problem. Consider a system consisting of n processes {P0, P1, ..., Pn−1}. Each process has a segment of code, called a critical section, in which the process may be changing common variables, updating a table, writing a file, and so on. The important feature of the system is that, when one process is executing in its critical section, no other process is allowed to execute in its critical section. That is, no two processes are executing in their critical sections at the same time. 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. The entry section and exit section are enclosed in boxes to highlight these important segments of code.

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.

PETERSON'S SOLUTION

Next, we illustrate a classic software-based solution to the critical-section problem known as Peterson's solution. Because of the way 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.

Peterson's solution is restricted to two processes that alternate execution between their critical sections and remainder sections. The processes are numbered P0 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.

To enter the critical section, process Pi first sets flag[i] to be true and then sets turn to the value there by 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. The eventual value of turn determines which of the two processes is allowed to enter its critical section first.

SYNCHRONIZATION HARDWARE

We have just described one software-based solution to the critical-section problem. However, as mentioned, software-based solutions such as Peterson's are not guaranteed to work on modern computer architectures. In the following discussions, we explore several more solutions to the critical-section problem using techniques ranging from hardware to software-based APIs available to both kernel developers and application programmers. All these solutions are based on the premise of locking —that is, protecting critical regions through the use of locks. As we shall see, the designs of such locks can be quite sophisticated.

This is often the approach taken by nonpreemptive
kernels. boolean test and set(boolean *target) {
boolean rv = *target;
*target =
true; return
rv;
}

do {
while (test and set(&lock))
; /* do nothing */
/* critical
section */ lock =
false;
/* remainder section */
} while (true);
Mutual-exclusion implementation with test and set().

Like the test and set() instruction, compare and swap() is
int compare and swap(int *value, int expected, int

```
new value) { int temp = *value;
if (*value == expected)
*value = new
value; return
temp;
}
```

SEMAPHORES

Mutex locks, as we mentioned earlier, are generally considered the simplest of synchronization tools. In this section, we examine a more robust tool that can behave similarly to a mutex lock but can also provide more sophisticated ways for processes to synchronize their activities.

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 verh ogen, "to increment"). The definition of wait() is as follows:

```
wait(S) {
while (S <=
0)
; // busy
wait 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 processmodifies 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 ($S \leq 0$), as well as its possible modification (S--), must be executed without interruption

CLASSIC PROBLEMS OF SYNCHRONIZATION

In this section, we present a number of synchronization problems as examples of a large class of concurrency- control problems. These problems are used for testing nearly every newly proposed synchronization scheme. In our solutions to the problems, we use semaphores for synchronization, since that is the traditional way to present such solutions. However, actual implementations of these solutions could use mutex locks in place of binary semaphores.

The Bounded-Buffer Problem

The bounded-buffer problem is commonly used to illustrate the power of synchronization primitives. Here, we present a general structure of this scheme without committing ourselves to any particular implementation

the producer and consumer processes share the following data structures:

```
int n;
semaphore mutex
= 1; semaphore
empty = n;
semaphore full =
0
```

We assume that the pool consists of n buffers, each capable of holding one item. The mutex semaphore provides mutual exclusion for accesses to the buffer pool and is initialized to the value 1. The empty and full semaphores count the number of empty and full buffers. The semaphore empty is initialized to the value n; the semaphore full is initialized to the value 0.

```
do {
wait(full);
wait(mutex)
;
. . .
/* remove an item from buffer to next consumed */
. . .
signal(mute
x);
signal(empt
y);
. . .
/* consume the item in next consumed */
. . .
} while (true);
```
The structure of the consumer process.

The Readers–Writers Problem

Suppose that a database is to be shared among several concurrent processes. Some of these processes may want only to read the database, whereas others may want to update (that is, to read and write) the database. We distinguish between these two types of processes by referring to the former as readers and to the latter as writers. Obviously, if two readers access the shared data simultaneously, no adverse effects will result. However, if a writer and some other process (either a reader or a writer) access the database simultaneously, chaos may ensue.

To ensure that these difficulties do not arise, we require that the writers have exclusive access to the shared database while writing to the database. This synchronization problem is referred to as the readers–writers problem

readers–writers problem, the reader processes share the following data structures:

```
semaphore rw mutex
= 1; semaphore
mutex = 1;
int read count = 0;

do {
wait(rw mutex);
. . .
/* writing is performed */
. . .
signal(rw mutex);
} while (true);
```
The structure of a writer process.

processes. The mutex semaphore is used to ensure mutual exclusion when the variable read count is updated.

MONITORS

Although semaphores provide a convenient and effective mechanism for process synchronization, using them incorrectly can result in timing errors that are difficult to detect, since these errors happen only if particular execution sequences take place and these sequences

do not always occur

To deal with such errors, researchers have developed high-level language constructs. In this section, we describe one fundamental high-level synchronization construct—the monitor type

```
monitor monitor name
{
/* shared variable
declarations */ function P1
( . . . ) {
. . .
}
function P2 ( . . . ) {
. . .
}
.
.
.
function Pn ( . . . ) {
. . .
}
initialization code ( . . . ) {
. . .
}
}
```

Syntax of a monitor.

Monitor Usage

An abstract data type—or ADT—encapsulates data with a set of functions to operate on that data that are independent of any specific implementation of the ADT. A monitor type is an ADT that includes a set of programmerdefined operations that are provided with mutual exclusion within the monitor. The monitor type also declares the variables whose values define the state of an instance of that type, along with the bodies of functions that operate on those variables. The syntax of a monitor type is above. The representation of a monitor type cannot be used directly by the various processes. Thus, a function defined within a monitor can access only those variables declared locally within the monitor and its formal parameters. Similarly, the local variables of a monitor can be accessed by only the local functions.

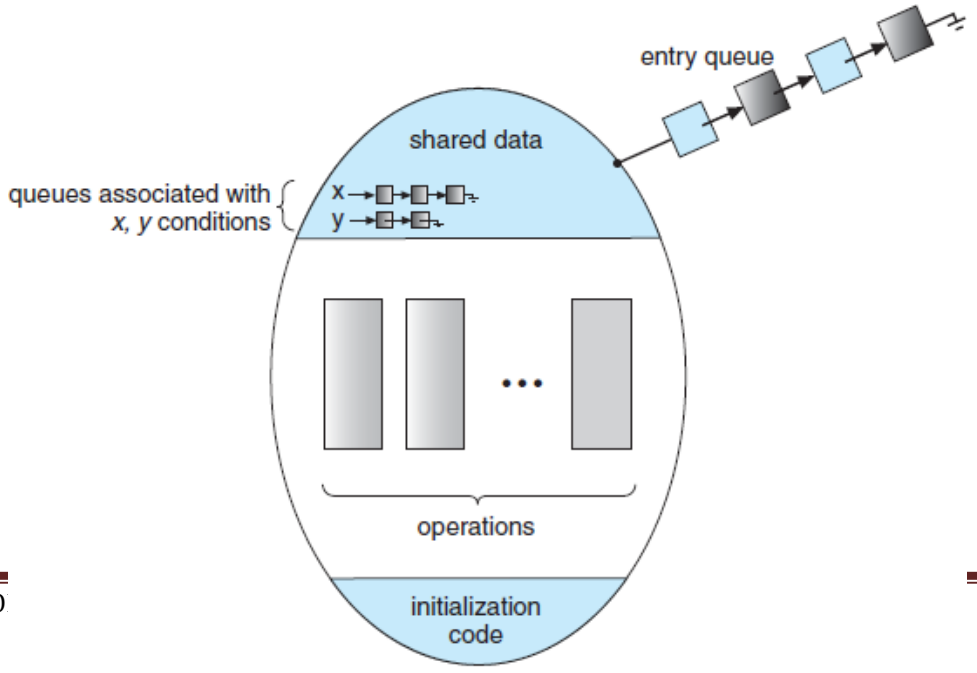


Figure 5.17 Monitor with condition variables.

Synchronization in Windows

TheWindows operating system is a multithreaded kernel that provides support for real-time applications and multiple processors. When theWindows kernel accesses a global resource on a single-processor system, it temporarily masks interrupts for all interrupt handlers that may also access the global resource. On a multiprocessor system, Windows protects access to global resources using spinlocks, although the kernel uses spinlocks only to protect short code segments. Furthermore, for reasons of efficiency, the kernel ensures that a thread will never be preempted while holding a spinlock. For thread synchronization outside the kernel, Windows provides dispatcher objects. Using a dispatcher object, threads synchronize according to several different mechanisms, includingmutex locks, semaphores, events, and timers. The system protects shared data by requiring a thread to gain ownership
of a mutex to access the data and to release ownership when it is finished. Semaphores behave as described in Section 5.6. Events are similar to condition variables; that is, they may notify a waiting thread when a desired condition occurs. Finally, timers are used to notify one (or more than one) thread that a specified amount of time has expired. Dispatcher objects may be in either a signaled state or a nonsignaled state. An object in a signaled state is available, and a thread will not block when acquiring the object. An object in a nonsignaled state is not available, and a thread will block when attempting to acquire the object.We illustrate the state transitions of a mutex lock dispatcher object.

A relationship exists between the state of a dispatcher object and the state of a thread. When a thread blocks on a nonsignaled dispatcher object, its state changes from ready to waiting, and the thread is placed in a waiting queue for that object. When the state for the dispatcher object moves to signaled, the kernel checks whether any threads are waiting on the object. If so, the kernel moves one thread—or possibly more—from the waiting state to the ready state, where they can resume executing. The number of threads the kernel selects from the waiting queue depends on the type of dispatcher object for which it is waiting. The kernel will select only one thread from the waiting queue for a mutex, since a mutex object may be "owned" by only a single thread. For an event object, the kernel will select all threads that are waiting for the event.

SYNCHRONIZATION IN LINUX

Prior to Version 2.6, Linux was a nonpreemptive kernel, meaning that a process running in kernel mode could not be preempted—even if a higher-priority process became available to run. Now, however, the Linux kernel is fully preemptive, so a task can be preempted when it is running in the kernel. Linux provides several different mechanisms for synchronization in the kernel. As most computer architectures provide instructions for atomic versions of simple math operations, the simplest synchronization technique within the Linux kernel is an atomic integer, which is represented using the opaque data type atomic t. As the name implies, all math operations using atomic integers are performed without interruption. The following code illustrates declaring an atomic integer counter and then performing various atomic operations:

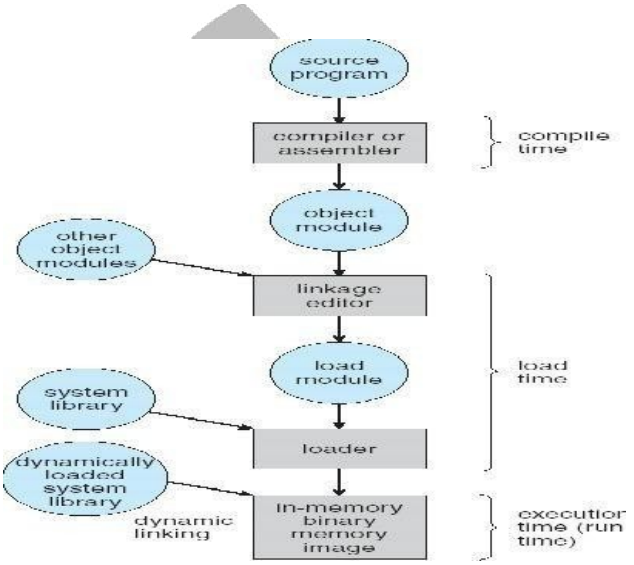
```
atomic t
counter; int
value;
atomic set(&counter,5); /* counter = 5 */
atomic add(10, &counter); /* counter =
counter + 10 */ atomic sub(4, &counter); /*
counter = counter - 4 */ atomic
inc(&counter); /* counter = counter + 1 */
value = atomic read(&counter); /* value = 12
*/
```

Atomic integers are particularly efficient in situations where an integer variable —such as a counter—needs to be updated, since atomic operations do not require the overhead of locking

mechanisms. However, their usage is limited to these sorts of scenarios. In situations where there are several variables contributing to a possible race condition, more sophisticated locking tools must be used.Mutex locks are available in Linux for protecting critical sections within the kernel. Here, a task must invoke the mutex lock() function prior to entering a critical section and the mutex unlock() function after exiting the critical section. If the mutex lock is unavailable, a task calling mutex lock() is put into a sleep state and is awakened when the lock's owner invokes mutex unlock().Linux also provides spinlocks and semaphores (as well as reader–writer versions of these two locks) for locking in the kernel. On SMP machines, the fundamental locking mechanism is a spinlock, and the kernel is designed so that the spinlock is held only for short durations. On single-processor machines, such as embedded systems with only a single processing core, spinlocks are inappropriate for use and are replaced by enabling and disabling kernel preemption. That is, on single-processor systems, rather than holding a spinlock, the kernel disables kernel preemption; and rather than releasing the spinlock, it enables kernel preemption.

UNIT - III:
Memory Management and Virtual Memory - Logical & physical Address Space, Swapping, Contiguous Allocation, Paging, Structure of Page Table. Segmentation, Segmentation with Paging, Virtual Memory, Demand Paging, Performance of Demanding Paging, Page Replacement - Page Replacement Algorithms, Allocation of Frames, Thrashing.

MEMORY MANAGEMENT:
In general, to rum a program, it must be brought into memory.
Input queue – collection of processes on the disk that are waiting to be brought into memory to run the program. User programs go through several steps before being run
Address binding: Mapping of instructions and data from one address to another address in memory. Three different stages of binding:
Compile time: Must generate absolute code if memory location is known in prior.
Load time: Must generate relocatable code if memory location is not known at compile time Execution time: Need hardware support for address maps (e.g., base and limit registers).



Logical vs. Physical Address Space

Logical address – generated by the CPU; also referred to as "virtual

address" Physical address – address seen by the memory unit.

Logical and physical addresses are the same in compile-time and load-time address-

binding schemes" Logical (virtual) and physical addresses differ in execution-time

address- binding scheme"

Memory-Management Unit (MMU)

It is a hardware device that maps virtual / Logical address to physical address

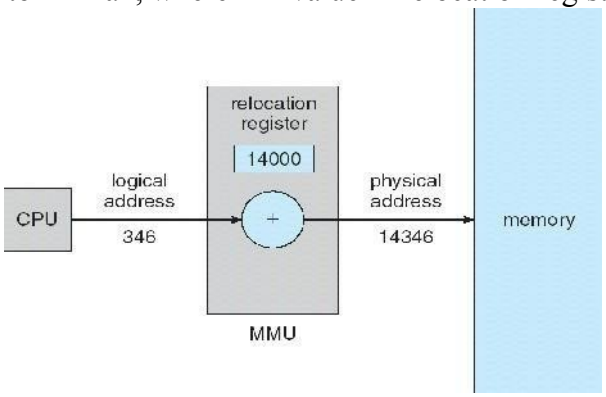
In this scheme, the relocation register's value is added to Logical address generated by

a user process. The user program deals with logical addresses; it never sees the real

physical addresses

Logical address range: 0 to max

Physical address range: R+0 to R+max, where R—value in relocation register.



Dynamic Loading

Through this, the routine is not loaded until it is called.
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 postponed until execution time & is particularly useful for libraries
Small piece of code called stub, used to locate the appropriate memory- resident library routine or function. Stub replaces itself with the address of the routine, and executes the routine
Operating system needed to check if routine is in processes' memory address
Shared libraries: Programs linked before the new library was installed will continue using

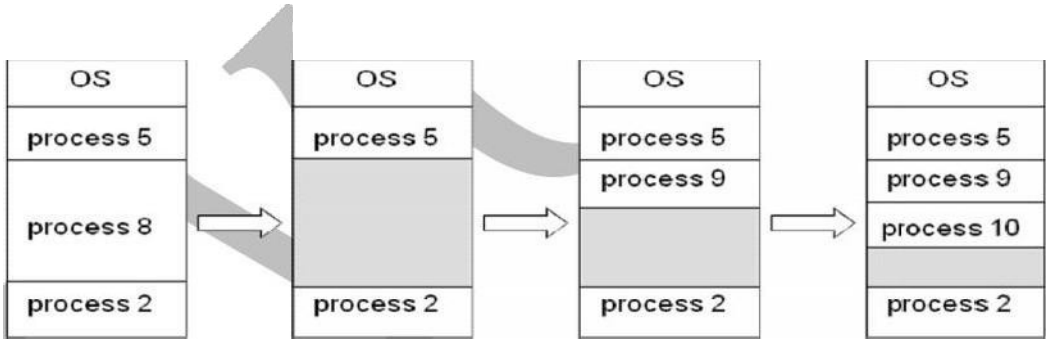
the older library. Swapping

A process can be swapped temporarily out of memory to a backing store (SWAP OUT)and then brought back into memory for continued execution (SWAP IN).
Backing store – fast disk large enough to accommodate copies of all memory images for all users & it must provide direct access to these memory images
Roll out, roll in – swapping variant used for priority-based scheduling algorithms; lower-priority process is swapped out so higher-priority process can be loaded and executed
Transfer time: Major part of swap time is transfer time. Total transfer time is directly proportional to the amount of memory swapped.
Example: Let us assume the user process is of size 1MB & the backing store is a standard hard disk with a transfer rate of 5MBPS.
Transfer time = 1000KB/5000KB per second
= 1/5 sec = 200ms

Contiguous Allocation
Each process is contained in a single contiguous section of memory. There are two methods namely:
Fixed – Partition

Method Variable –
Partition Method Fixed
– Partition Method :
Divide memory into fixed size partitions, where each partition has exactly one process.
The drawback is memory space unused within a partition is
wasted.(eg.when process size < partition size)
Variable-partition method:
Divide memory into variable size partitions, depending upon the size of the
incoming process. When a process terminates, the partition becomes available for
another process.
As processes complete and leave they create holes in the main memory.

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



memory. Dynamic Storage- Allocation Problem:

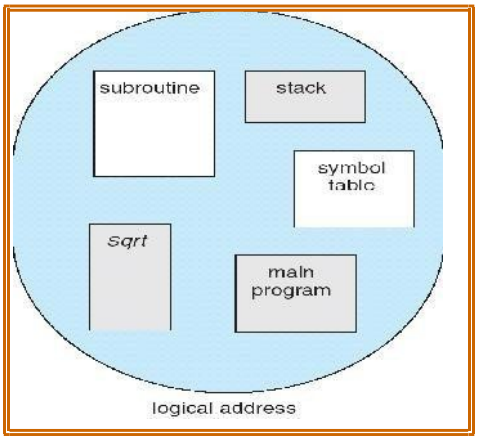
How to satisfy a request of size n‘ from a list of free

holes? Solution:

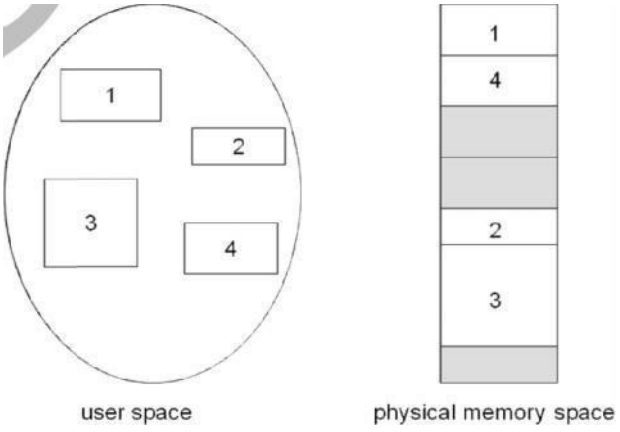
First-fit: Allocate the first hole that is big enough.
Best-fit: Allocate the smallest hole that is big enough; must search entire list, unless ordered by size.
Produces the smallest leftover hole.
Worst-fit: Allocate the largest hole; must also search entire list. Produces the
largest leftover hole.

NOTE: First-fit and best-fit are better than worst-fit in terms of speed and
storage utilization Fragmentation:
o External Fragmentation – This takes place when enough total memory space exists to satisfy a request, but it is not contiguous i.e, storage is fragmented into a large number of small holes scattered throughout the main memory.
o Internal Fragmentation – Allocated memory may be slightly larger than
requested memory. Example: hole = 184 bytes Process size = 182 bytes.
We are left with a hole of 2
bytes. o Solutions
Coalescing: Merge the adjacent holes together.
Compaction: Move all processes towards one end of memory, hole towards other end
of memory, producing one large hole of available memory. This scheme is expensive as it
can be done if relocation is dynamic and done at execution time.
Permit the logical address space of a process to be non-contiguous. This is achieved
through two memory management schemes namely paging and segmentation.
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, arrays



Logical View of Segmentation



Segmentation Hardware

o Logical address consists of a two tuple : <Segment-number, offset>

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

- □ **Base** – contains the starting physical address where the segments reside in memory
- □ **Limit** – specifies the length of the segment

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

o ***Segment-table length register (STLR)*** indicates number of segments used by a program; Segment number=s' is legal, if *s* < STLR o **Relocation**.

- □ dynamic
- □ by segment

table o **Sharing**.

- □ shared segments
- □ same segment

number o **Allocation**.

- □ first fit/best fit
- □ external fragmentation

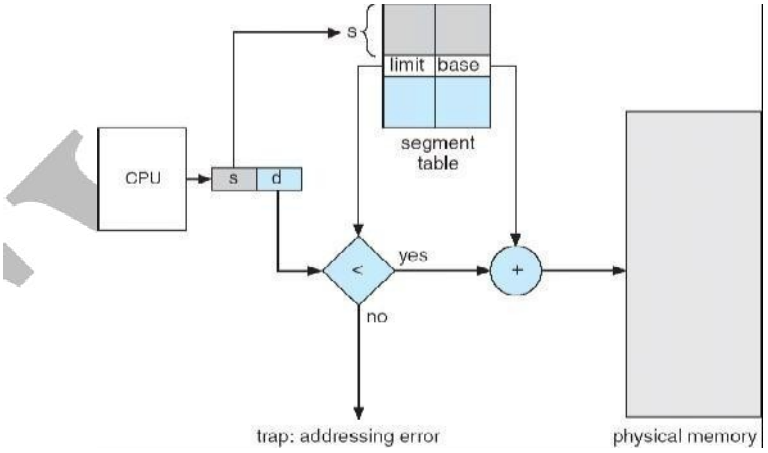
o **Protection:** With each entry in segment table associate:

- □ validation bit = 0 □ illegalsegment
- □ read/write/execute privileges

o Protection bits associated with segments; code sharing occurs at segment level

o Since segments vary in length, memory allocation is a dynamic storage- allocation problem

o A segmentation example is shown in the following diagram



EXAMPLE:

o Another advantage of segmentation involves the sharing of code or data.

o Each process has a segment table associated with it, which the dispatcher uses to define the hardware segment table when this process is given the CPU.

o Segments are shared when entries in the segment tables of two different processes point to the same physical location.

Segmentation with paging

o The IBM OS/ 2.32 bit version is an operating system running on top of the Intel 386 architecture. The 386 uses segmentation with paging for memory management. The maximum number of segments per process is 16 KB, and each segment can be as large as 4 gigabytes.

o The local-address space of a process is divided into two partitions.

- □ The first partition consists of up to 8 KB segments that are private to that process.
- □ The second partition consists of up to 8KB segments that are shared among all the processes.

o Information about the first partition is kept in the **local descriptor table (LDT)**, information about the second partition is kept in the **global descriptor table (GDT)**.

- o Each entry in the LDT and GDT consist of 8 bytes, with detailed information about a particular segment including the base location and length of the segment.

The logical address is a pair (selector, offset) where the selector is a16-bit number:

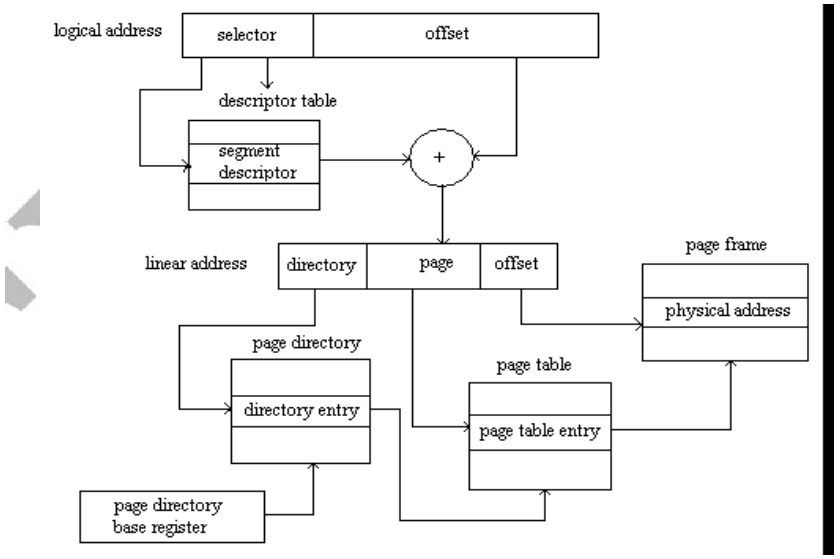
s	g	p
13	**1**	**2**

Where s designates the segment number, g indicates whether the segment is in the GDT or LDT, and p deals with protection. The offset is a 32-bit number specifying the location of the byte within the segment in question.

- The base and limit information about the segment in question are used to generate a linear- address.
- First, the limit is used to check for address validity. If the address is not valid, a memory fault is generated, resulting in a trap to the operating system. If it is valid, then the value of the offset is added to the value of the base, resulting in a 32-bit linear address. This address is then translated into a physical address.
- The linear address is divided into a page number consisting of 20 bits, and a page offset consisting of 12 bits. Since we page the page table, the page number is further divided into a 10-bit page directory pointer and a 10-bit

  page table pointer. The logical address is as follows.

p_1	p_2	d
10	**10**	**12**



o To improve the efficiency of physical memory use. Intel 386 page tables can be swapped to disk. Inthis case, an invalid bit is used in the page directory entry to indicate whether the table to which the entry is pointing is in memory or on disk.

o If the table is on disk, the operating system can use the other 31 bits to specify the disk location of thetable; the table then can be brought into memory on demand.

Paging

- It is a memory management scheme that permits the physical address space of a process to be noncontiguous.
- It avoids the considerable problem of fitting the varying size memory chunks on to the backing store.

(i) Basic Method:

o Divide logical memory into blocks of same size called **"pages"**. o Divide physical memory into fixed-sized blocks called **"frames"** o Page size is a power of 2, between 512 bytes and 16MB.

Address Translation Scheme

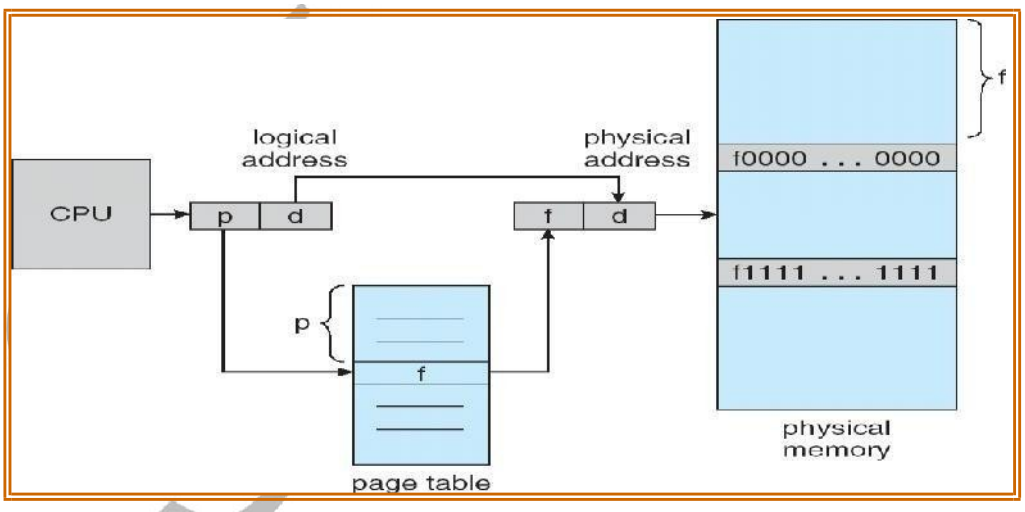
o Address generated by CPU(logical address) is divided into:

□ **Page number *(p)*** – used as an index into a page table which contains base address of each page in physical memory

□ **Page offset *(d)*** –combined with base address to define the physical address i.e.,

Physical address = base address + offset

Paging Hardware



Demand Paging

- It is similar to a paging system with swapping.
- Demand Paging - Bring a page into memory only when it is needed
- To execute a process, swap that entire process into memory.
  Rather than swapping theentireprocessintomemoryhowever,
  we use Lazy Swapper"
- **Lazy Swapper** - Never swaps a page into memory unless
  that page will be needed.
- **Advantages**
  - Less I/O needed
  - Less memory needed
  - Faster response
  - More users

Page Replacement

- If no frames are free, we could find one that is not currently
  being used & free it.
- We can free a frame by writing its contents to swap space & changing the page
  table to indicate that the page is no longer in memory.
- Then we can use that freed frame to hold the page for which the process faulted.

Basic Page Replacement

1. Find the location of the desired page on disk
2. Find a free frame
   - If there is a free frame , then use it.
   - If there is no free frame, use a page replacement algorithm to select a **victim** frame
   - Write the victim page to the disk, change the page & frame tables accordingly.
3. Read the desired page into the (new) free frame. Update the page and frame tables.

UNIT - IV:

File System Interface - The Concept of a File, Access methods, Directory Structure, File System Mounting, File Sharing, Protection, File System Implementation - File System Structure, File System Implementation, Allocation methods, Free-space Management, Directory Implementation, Efficiency and Performance.

Mass Storage Structure - Overview of Mass Storage Structure, Disk Structure, Disk Attachment, Disk Scheduling, Disk Management, Swap space Management.

File System Storage-File Concepts

File Concept

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

- From a user's perspective, a file is the smallest allotment of logical secondary storage; that

is, data cannot be written to secondary storage unless they are within a file.

Examples of files:

- A text file is a sequence of characters organized into lines (and possibly pages). A source file is a sequence of subroutines and functions, each of which is further organized as declarations followed by executable statements. An object file is a sequence of bytes organized into blocks understandable by the system's linker.

An executable file is a series of code sections that the loader can bring into memory and execute.

File Attributes

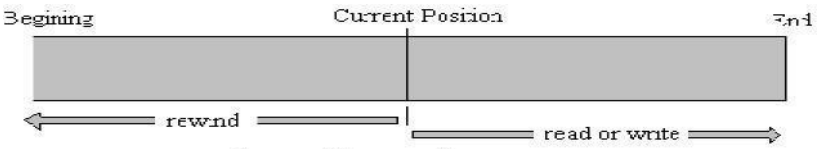
- **Name:** The symbolic file name is the only information kept in human readable form.
- **Identifier:** This unique tag, usually a number identifies the file within the file system. It is the non-human readable name for the file.
- **Type:** This information is needed for those systems that support different types.
- **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 sizeare 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.

File Operations

- Creating a file
- Writing a file
- Reading a file
- Repositioning within a file
- Deleting a file
- Truncating a file

Access Methods

1. **Sequential Access**
   a. The simplest access method is sequential access. Information in the file is processed in order, one record after the other. This mode of access is by far the most common; for example, editors and compilers usually access files in this fashion.



F:g 4.10 Sequential-access file

• The bulk of the operations on a file is reads and writes. A read operation reads the next portion of the file and automatically advances a file pointer, which tracks the I/O location. Similarly, a write appends to the end of the file and advances to the end of the newly written material (the new end of file). Such a file can be reset to the beginning and, on some systems, a program may be able to skip forward or back ward n records, for some integer n-perhaps only for n=1. Sequential access is based on a tape model of a file, and works as well on sequential-access devices as it does on random – access ones.
•

2. Direct Access

• Another method is direct access (or relative access). A file is made up of fixed length logical records that allow programs to read and write records rapidly in no particular order. The direct- access methods is based on a disk model of a file, since disks allow random access to any file block.
•
• For direct access, the file is viewed as a numbered sequence of blocks or records. A direct-access file allows arbitrary blocks to be read or written. Thus, we may read block 14, then read block 53, and then write block7. There are no restrictions on the order of reading or writing for a direct-access file.
•
• Direct – access files are of great use for immediate access to large amounts of information. Database is often of this type. When a query concerning a particular subject arrives, we compute which block contains the answer, and then read that block directly to provide the desired information.

• **Directory and Disk Structure**

<u>There are five directory structures. They are</u>

1. Single-level directory
2. Two-level directory
3. Tree-Structured directory
4. Acyclic Graph directory
5. General Graph directory

1. Single – Level Directory

- The simplest directory structure is the single- level directory.
- All files are contained in the same directory.

- **Disadvantage:**
  - ➢ When the number of files increases or when the system has more than one user, since all filesare in the same directory, they must have unique names.

2. Two – Level Directory

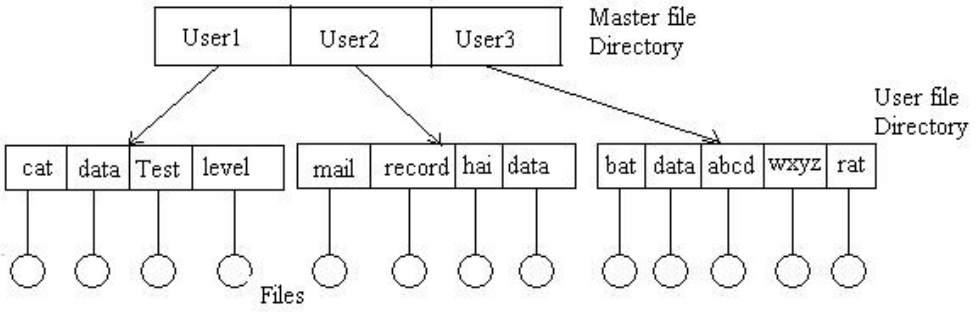
- In the two level directory structures, each user has her own user file directory (UFD).
- When a user job starts or a user logs in, the system's master file directory (MFD) is searched. The MFDis

indexed by user name or account number, and each entry points to the UFD for that user.

- When a user refers to a particular file, only his own UFD is searched.
- Thus, different users may have files with the same name.
- Although the two – level directory structure solves the name-collision problem

Disadvantage:

➢ Users cannot create their own sub-directories.



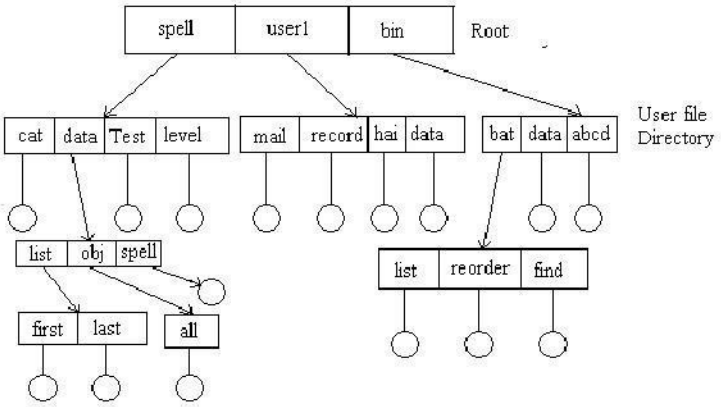
3. Tree – Structured Directory

- A tree is the most common directory structure.
- The tree has a root directory. Every file in the system has a unique path name.
- A **path name** is the path from the root, through all the subdirectories to a specified file.
- A directory (or sub directory) contains a set of files or sub directories.
- A directory is simply another file. But it is treated in a special way.
- All directories have the same internal format.

- One bit in each directory entry defines the entry as a file (0) or as a subdirectory(1).
- Special system calls are used to create and delete directories.
- Path names can be of two types: absolute path names or relative path names.
- An absolute path name begins at the root and follows a path down to the specified file, giving the directory

names on the path.

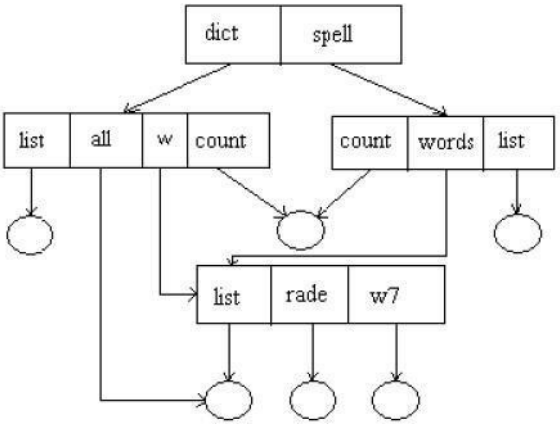
- A relative path name defines a path from the current directory.



4. Acyclic Graph Directory.

- An acyclic graph is a graph with no cycles.
- To implement shared files and subdirectories this directory structure is used.
- An acyclic – graph directory structure is more flexible than is a simple tree structure, but it is also more complex. In a system where sharing is implemented by symbolic link, this situation is somewhat easier to handle. The deletion of a link does not need to affect the original file; only the link is removed.
- Another approach to deletion is to preserve the file until all references to it are deleted. To implementthis

approach, we must have some mechanism for determining that the last reference to the file has been deleted.



1. Multiple Users:

- When an operating system accommodates multiple users, the issues of file sharing, file naming and file

protection become preeminent.

- The system either can allow user to access the file of other users by default, or it may require that a user specifically grant access to the files.
- These are the issues of access control and protection.
- To implementing sharing and protection, the system must maintain more file and directory attributes than a on a single-user system.
- The owner is the user who may change attributes, grand access, and has the most control over the file or

directory.

- The group attribute of a file is used to define a subset of users who may share access to the file.
- Most systems implement owner attributes by managing a list of user names and associated user identifiers

(user Ids).

- When a user logs in to the system, the authentication stage determines the appropriate user ID for the user. That user ID is associated with all of user's processes and threads. When they need to be user readable, they are translated, back to the user name via the user name list. Likewise, group functionality can be implemented as a system wide list of group names and group identifiers.
- Every user can be in one or more groups, depending upon operating system design decisions. The user's

group Ids is also included in every associated process and thread.

2. Remote File System:

- Networks allowed communications between remote computers.
- Networking allows the sharing or resource spread within a campus or even around the world.
- User manually transfer files between machines via programs like **ftp**.
- A **distributed file system** (DFS) in which remote directories is visible from the local machine.
- The **World Wide Web**: A browser is needed to gain access to the remote file and separate operations (essentially a wrapper for ftp) are used to transfer files.

a) The client-server Model:

- Remote file systems allow a computer to a mount one or more file systems from one ormore remote machines.
- A server can serve multiple clients, and a client can use multiple servers, depending on the

implementation details of a given client –server facility.

- Client identification is more difficult. Clients can be specified by their network name or other identifier, such as IP address, but these can be spoofed (or imitate). An unauthorized client can spoof the server into deciding that it is authorized, and the unauthorized client could be allowed access.

b) Distributed Information systems:

- Distributed information systems, also known as distributed naming service,

have been devisedto
provide a unified access to the information needed for remote computing.

- Domain name system (DNS) provides host-name-to-network address translations for theirentire Internet (including the World Wide Web).
- Before DNS was invented and became widespread, files containing the same information were sent

via e-mail of ftp between all networked hosts.

c) Failure Modes:

- **Redundant arrays of inexpensive disks** (**RAID**) can prevent the loss of a disk from resulting in the loss of data.
- Remote file system has more failure modes. By nature of the complexity of networking system and the required interactions between remote machines, many more problems can interfere with the proper operation of remote file systems.

d) Consistency Semantics:

- It is characterization of the system that specifies the semantics of multiple users accessing ashared file

simultaneously.

- These semantics should specify when modifications of data by one user are observable by other users.
- The semantics are typically implemented as code with the file system.
- A series of file accesses (that is reads and writes) attempted by a user to the same file is always enclosed between the open and close operations.
- The series of access between the open and close operations is a **file session**.

(i) UNIX Semantics:

The UNIX file system uses the following consistency semantics:

1. Writes to an open file by a user are visible immediately to other users that have this file open at thesame time.
2. One mode of sharing allows users to share the pointer of current location into the file. Thus, the advancing of the pointer by one user affects all sharing users.

(ii) Session Semantics:

The Andrew file system (AFS) uses the following consistency semantics:

1. Writes to an open file by a user are not visible immediately to other users that have the same file open simultaneously.
2. Once a file is closed, the changes made to it are visible only in sessions starting later. Already open instances of the file do not reflect this change.

(iii) Immutable –shared File Semantics:

- Once a file is declared as shared by its creator, it cannot be modified.
- An immutable file has two key properties:

 Its name may not be reused and its contents may not be altered.

File Protection

(i) Need for file protection.

- When information is kept in a computer system, we want to keep it safe from **physical damage** (reliability) and **improper access** (protection).
- Reliability is generally provided by duplicate copies of files. Many computers have

systems programs that automatically (or though computer-operator intervention) copy disk files to tape at regular intervals (once per day or week or month) to maintain a copy should a file system be accidentally destroyed.

• File systems can be damaged by hardware problems (such as errors in reading or writing), power surges or failures, head crashes, dirt, temperature extremes, and vandalism. Files may be deleted accidentally. Bugs in the file-system software can also cause file contents to be lost.

• Protection can be provided in many ways. For a small single-user system, we might provide protectionby physically removing the floppy disks and locking them in a desk drawer or file cabinet. In a multi-user system, however, other mechanisms are needed.

(ii) Types of Access

• Complete protection is provided by prohibiting access.

• Free access is provided with no protection.

• Both approaches are too extreme for general use.

• What is needed is **controlled access**.

• Protection mechanisms provide controlled access by limiting the types of file access that can be made. Access is permitted or denied depending on several factors, one of which is the type of access requested. Several different types of operations may be controlled:

1. **Read:** Read from the file.
2. **Write:** Write or rewrite the file.
3. **Execute:** Load the file into memory and execute it.
4. **Append:** Write new information at the end of the file.
5. **Delete:** Delete the file and free its space for possible reuse.
6. **List:** List the name and attributes of the file.

(iii) Access Control

• Associate with each file and directory an access-control list (ACL) specifying the user name and thetypes of access allowed for each user.

• When a user requests access to a particular file, the operating system checks the access list associated with that file. If that user is listed for the requested access, the access is allowed. Otherwise, a protection violation occurs and the user job is denied access to the file.

• This technique has two undesirable consequences:

 Constructingsuchalistmaybeatediousandunrewardingtask,especiallyifwedonotknowinadvance **the list of users in the system.**

 Thedirectoryentry,previouslyoffixedsize,nowneedstobeofvariablesize,resultinginmore **complicated space management.**

• To condense the length of the access control list, many systems recognize three classifications of usersin **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.

File System Implementation- File System Structure

- **Disk** provide the bulk of secondary storage on which a file system is maintained.
- **Characteristics of a disk:**

1. They can be rewritten in place, it is possible to read a block from the disk, to modify the block and towrite it back into the same place.
2. They can access directly any given block of information to the disk.

- To produce an efficient and convenient access to the disk, the operating system imposes one or more file system to allow the data to be stored, located and retrieved easily.
- The file system itself is generally composed of many different levels. Each level in the design uses the

features of lower level to create new features for use by higher levels.

Layered File System

- The **I/O control** consists of device drivers and interrupt handlers to transfer information between themain memory and the disk system .
- The **basic file system** needs only to issue generic commands to the appropriate device driver to read and write physical blocks on the disk. Each physical block is identified by its numeric disk address (forexample, drive –1, cylinder 73, track 2, sector 10)

Directory Implementation

1. Linear List

- The simplest method of implementing a directory is to use a linear list of file names with pointer to the data

blocks.

- A linear list of directory entries requires a linear search to find a particular entry.
- This method is simple to program but time- consuming to execute. To create a new file, we must first search the but time – consuming to execute.
- The real disadvantage of a linear list of directory entries is the linear search to find a file.

2. Hash Table

- In 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 thelinear

list.

- Therefore, it can greatly decrease the directory search time.
- Insertion and deleting are also fairly straight forward, although some provision must be made for collisions

– situation where two file names hash to the same location.

- The major difficulties with a hash table are its generally fixed size and the dependence ofthe hash function on that size.

Allocation Methods

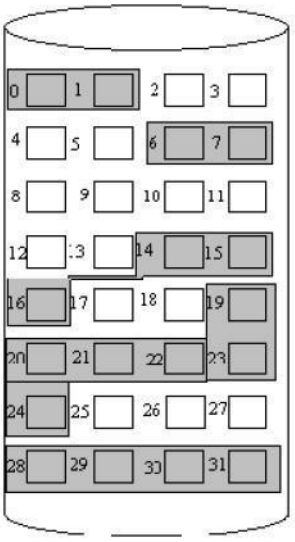
- The main problem is how to allocate space to these files so that disk space is utilized effectively and files

can be accessed quickly.

- There are three major methods of allocating disk space:
  1. Contiguous Allocation
  2. Linked Allocation
  3. Indexed Allocation

1. Contiguous Allocation

- The contiguous – allocation method requires each file to occupy a set of contiguous blocks on



Directory

file	start	length
Count	9	2
tr	14	3
mail	19	5
list	28	4
f	6	2

the disk.

- Contiguous allocation of a file is defined by the disk address and length (in block units) of the first block. If the file is n blocks long and starts at location b, then it occupies blocks b,. b+1, b+2,….,b+n-1.
- The directory entry for each file indicates the address of the starting block and the length of the area allocated for this file.

Disadvantages:

1. Finding space for a new file.

- The contiguous disk space-allocation problem suffer from the problem of external fragmentation. As file are allocated and deleted, the free disk space is broken into chunks. It becomes a problem when the largest contiguous chunk is insufficient for a request; storage is fragmented into a number of holes, no one of which is large enough to store the data.

2. Determining how much space is needed for a file.

- When the file is created, the total amount of space it will need must be found an allocated how does the

creator know the size of the file to be created?

- If we allocate too little space to a file, we may find that file cannot be extended. The other possibility is to find a larger hole, copy the contents of the file to the new space, and release the previous space. This series of actions may be repeated as long as space exists, although it can be time – consuming. However, in this case, the user never needs

to be informed explicitly about what is happening ; the system continues despite the problem, although more and more slowly.

- Even if the total amount of space needed for a file is known in advance pre-allocation may be inefficient.
- A file that grows slowly over a long period (months or years) must be allocated enough space for its final size, even though much of that space may be unused for a long time the file, therefore has a large amount of internal fragmentation.

To overcome these disadvantages:

- Use a modified contiguous allocation scheme, in which a contiguous chunk of space called as an **extent** is allocated initially and then, when that amount is not large enough another chunk of contiguous space an extent is added to the initial allocation.
- Internal fragmentation can still be a problem if the extents are too large, and external fragmentation can be a problem as extents of varying sizes are allocated and deallocated.

2. Linked Allocation

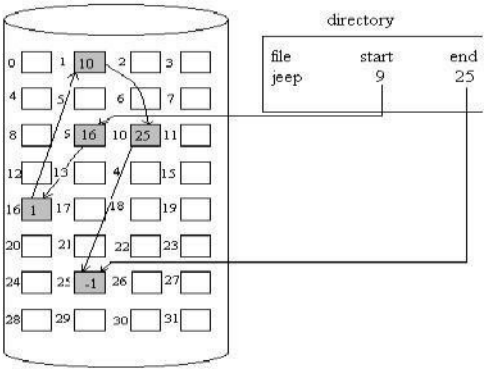
- Linked allocation solves all problems of contiguous allocation.
- With linked allocation, each file is a linked list of disk blocks, the disk blocks may be scattered any where on the disk.
- The directory contains a pointer to the first and last blocks of the file. For example, a file of five blocks

might start at block 9, continue at block 16, then block 1, block 10, and finally bock 25.

- Each block contains a pointer to the next block. These pointers are not made available to the user.
- There is no external fragmentation with linked allocation, and any free block on the free space list can be

used to satisfy a request.

- The size of a file does not need to the declared when that file is created. A file can continue to grow as long as free blocks are available consequently, it is never necessary



to compacts disk space.

UNIT - V:

Deadlocks - System Model, Deadlock Characterization, Methods for Handling Deadlocks, Deadlock Prevention, Deadlock Avoidance, Deadlock Detection, Recovery from Deadlock.

Protection - System Protection, Goals of Protection, Principles of Protection, Domain of Protection, Access Matrix, Implementation of Access Matrix, Access Control, Revocation of Access Rights, Capability-Based Systems, Language-Based Protection.

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

SYSTEM MODEL:

A system consists of a finite number of resources to be distributed among a number of competing processes. The resources may be partitioned into several types (or classes), each consisting of some number of identical instances. CPU cycles, files, and I/O devices (such as printers and DVD drives) are examples of resource types. If a system has two CPUs, then the resource type CPU has two instances. Similarly, the resource type printer may have five instances.

A process may utilize a resource in only the following sequence:

1. **Request**. The process requests the resource. If the request cannot be granted immediately (for example, if the resource is being used by another process), then the requesting process must wait until it can acquire the resource.
2. **Use**. The process can operate on the resource (for example, if the resource is a printer, the process can print on the printer).
3. **Release**. The process releases the resource.

DEADLOCK CHARACTERIZATION

In a deadlock, processes never finish executing, and system resources are tied up, preventing other jobs fromstarting.

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

1. **Mutual exclusion**. At least one resource must be held in a non-sharable mode; that is, only one process at a time can use the resource. If another process requests that resource, the requesting process must be delayed until the resource has been released.
2. **Hold and wait**. A process must be holding at least one resource and waiting to acquire additional resources that are currently being held by other processes.

3. **No preemption**. Resources cannot be preempted; that is, a resource can be released only voluntarily by the process holding it, after that process has completed its task.
4. **Circular wait**. A set *{P*0, *P*1, ..., *Pn}* of waiting processes must exist such that *P*0 is waiting for a resource held by *P*1, *P*1 is waiting for a resource held by *P*2, ..., *Pn*−1 is waiting for a resource held by *Pn*, and *Pn* is waiting for a resource held by *P*0.

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

METHODS FOR HANDLING DEADLOCKS

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

- We can use a protocol to prevent or avoid deadlocks, ensuring that the system will never enter a deadlocked state.
- We can allow the system to enter a deadlocked state, detect it, and recover.
- We can ignore the problem altogether and pretend that deadlocks never occur in the system.

DEADLOCK PREVENTION

Deadlock prevention provides a set of methods to ensure that at least one of the necessary conditions cannot hold.

Mutual Exclusion

The mutual exclusion condition must hold. That is, at least one resource must be nonsharable. Sharable resources, in contrast, do not require mutually exclusive access and thus cannot be involved in a deadlock. Read-only files are a good example of a sharable resource.

Hold and Wait

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

No Preemption

The third necessary condition for deadlocks is that there be no preemption of resources that have already been allocated. To ensure that this condition does not hold, we can use the following protocol. If a process is holding some resources and requests another resource that cannot be immediately allocated to it (that is, the process must wait), then all resources the process is currently holding are preempted

Circular Wait

The fourth and final condition for deadlocks is the circular-wait condition. One way to ensure that this condition never holds is to impose a total ordering of all resource types and to require that each process requests resources in an increasing order of enumeration

DEADLOCK AVOIDANCE

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

Safe State

A state is safe if the system can allocate resources to each process (up to its maximum) in some order and still avoid a deadlock. More formally, a system is in a safe state only if there exists a

safe sequence. A sequence of processes
<P1, P2, ..., Pn> is a safe sequence for the current allocation state if, for each Pi , the resource requests that Pi can still make can be satisfied by the currently available resources plus the resources held by all Pj, with j < i. In this situation, if the resources that Pi needs are not immediately available, then Pi can wait until all Pj have finished. When they have finished, Pi can obtain all of its needed resources, complete its designated task, return its allocated resources, and terminate. When Pi terminates, Pi+1 can obtain its needed resources, and so on. If no such sequence exists, then the system state is said to be unsafe

Resource-Allocation-Graph Algorithm
If we have a resource-allocation system with only one instance of each resource type, we can use a variant of the resource-allocation graph defined for deadlock avoidance. In addition to the request and assignment edges already described, we introduce a new type of edge, called a claim edge. A claim edge Pi → Rj indicates that process Pi may request resource Rj at some time in the future. This edge resembles a request edge in direction but is represented in the graph by a dashed line. When process Pi requests resource Rj , the claim edge Pi → Rj is converted to a request edge. Similarly, when a resource Rj is released by Pi , the assignment edge Rj → Pi is reconverted to a claim edge Pi → Rj

BANKER'S ALGORITHM

The resource-allocation-graph algorithm is not applicable to a resourceallocation system with multiple instances of each resource type. The deadlockavoidance algorithm that we describe next is applicable to such a system but is less efficient than the resource-allocation graph scheme. This algorithm is commonly known as the banker's algorithm

DEADLOCK DETECTION

If a system does not employ either a deadlock-prevention or a deadlockavoidance algorithm, then a deadlock situation may occur. In this environment, the system may provide:
• An algorithm that examines the state of the system to determine whether a deadlock has occurred
• An algorithm to recover from the deadlock

RECOVERY FROM DEADLOCK

When a detection algorithm determines that a deadlock exists, several alternatives are available. One possibility is to inform the operator that a deadlock has occurred and to let the operator deal with the deadlock manually. Another possibility is to let the system recover from the deadlock automatically.

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 great expense. The deadlocked processes may have computed for a long time, and the results of these partial computations must be discarded and probably will have to be 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.

PROTECTION

Protection refers to a mechanism for controlling the access of programs, processes, or users to the resources defined by a computer system.

GOALS OF PROTECTION

Protection can improve reliability by detecting latent errors at the interfaces between component subsystems. 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.

PRINCIPLES OF PROTECTION

a guiding principle can be used throughout a project, such as the design of an operating system. Following this principle simplifies design decisions and keeps the system consistent and easy to understand. 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.

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.

Domain Structure

To facilitate the scheme just described, a process operates within 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 access right.

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

IMPLEMENTATION OF THE ACCESS MATRIX

Global Table

The simplest implementation of the access matrix is a global table consisting of a set of ordered triples <domain, object, rights-set>. Whenever an operation M is executed on an object Oj within domain Di , the global table
is searched for a triple <Di , Oj , Rk>, with $M \in Rk$. If this triple is found, the operation is allowed to continue; otherwise, an exception (or error) condition is raised.

Access Lists for Objects

Each column in the access matrix can be implemented as an access list for one object, as described in Obviously, the empty entries can be discarded. The resulting list for each object consists of ordered pairs <domain, rights-set>, which define all domains with a nonempty set of access rights for that object.

Capability Lists for Domains

Rather than associating the columns of the access matrix with the objects as access lists, we can associate each row with its domain. A capability list for a domain is a list of objects together with the operations allowed on those objects. An object is often represented by its physical

name or address, called a capability. To execute operation M on object Oj , the process executes the operation M, specifying the capability (or pointer) for object Oj as a parameter. Simple possession of the capability means that access is allowed.

ACCESS CONTROL

Each file and directory is assigned an owner, a group, or possibly a list of users, and for each of those entities, access-control information is assigned. A similar function can be added to other aspects of a computer system. A good example of this is found in Solaris 10. Solaris 10 advances the protection available in the operating system by explicitly adding the principle of least privilege via role-based access control (RBAC).

REVOCATION OF ACCESS RIGHTS

In a dynamic protection system, we may sometimes need to revoke access rights to objects shared by different users. Various questions about revocation may arise:

- Immediate versus delayed. Does revocation occur immediately, or is it delayed? If revocation is delayed, can we find out when it will take place?
- Selective versus general. When an access right to an object is revoked, does it affect all the users who have an access right to that object, or can we specify a select group of users whose access rights should be revoked?
- Partial versus total. Can a subset of the rights associated with an object be revoked, or must we revoke all access rights for this object?
- Temporary versus permanent. Can access be revoked permanently (that is, the revoked access right will never again be available), or can access be revoked and later be obtained again?

Capability-Based Systems

In this section, we survey two capability-based protection systems. These systems differ in their complexity and in the types of policies that can be implemented on them.

An Example: Hydra

Hydra is a capability-based protection system that provides considerable flexibility. The system implements a fixed set of possible access rights, including such basic forms of access as the right to read, write, or execute a memory segment. In addition, a user (of the protection system) can declare other rights. The interpretation of user-defined rights is performed solely by the user's program, but the system provides access protection for the use of these rights, as well as for the use of system-defined rights. These facilities constitute a significant development in protection technology

An Example: Cambridge CAP System

A different approach to capability-based protection has been taken in the design of the Cambridge CAP system. CAP's capability system is simpler and superficially less powerful than that of Hydra. However, closer examination shows that it, too, can be used to provide secure protection of user-defined objects. CAP has two kinds of capabilities. The ordinary kind is called a data capability. It can be used to provide access to objects, but the only rights provided are the standard read, write, and execute of the individual storage segments associated with the object. Data capabilities are interpreted by microcode in the CAP machine.

The second kind of capability is the so-called software capability, which is protected, but not interpreted, by the CAP microcode. It is interpreted by a protected (that is, privileged) procedure, which may be written by an application programmer as part of a subsystem.

LANGUAGE-BASED PROTECTION

The designers of protection systems have drawn heavily on ideas that originated in programming languages and especially on the concepts of abstract data types and objects

Compiler-Based Enforcement

A variety of techniques can be provided by a programming-language implementation to enforce protection, but any of these must depend on some degree of support from an underlying machine and its operating system. For example, suppose a language is used to generate code to run on the Cambridge CAP system. On this system, every storage reference made on the underlying hardware occurs indirectly through a capability. This restriction prevents any process from accessing a resource outside of its protection environment at any time.

Protection in Java

Because Java was designed to run in a distributed environment, the Java virtual machine—or JVM—has many built- in protection mechanisms. Java programs are composed of classes, each of which is a collection of data fields and functions (called methods) that operate on those fields. The JVM loads a class in response to a request to create instances (or objects) of that class. One of the most novel and useful features of Java is its support for dynamically loading untrusted classes over a network and for executing mutually distrusting classes within the same JVM.